Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria - DEIB
Doctoral Program in Information Technology

From Theoretical To Real World Cryptography: Towards Practical Privacy-Preserving Outsourced Computation and Accurate Parsing of Digital Certificates

Doctoral Dissertation of:
Nicholas Mainardi

Supervisor:
Prof. Gerardo Pelosi

Tutor:
Prof. Cristina Silvano

Chair of the Doctoral Program:
Prof. Barbara Pernici

2020 – XXXII Cycle

Acknowledgments

The most great and precious contribution to this research is certainly due to my advisor Prof. Gerardo Pelosi, who has always been strongly committed to guide me throughout all my PhD. I am grateful for all his patience and the time he devoted to show me how to describe a scientific work in a clear and rigorous manner, especially for the headaches I often gave him when reading the drafts of our papers; to be honest, it was sometimes tedious and frustrating to see entire paragraphs being rewritten, but I definitely acknowledge the great improvements to my scientific writing skills thanks to his advice and the many important details making a scientific paper pleasant to read that I learned from him. I am also thankful for the valuable feedback and insights on my research activity, which significantly improve the quality of our works. Another pillar of this research is definitely Prof. Alessandro Barenghi, who contributed to all the works reported in this thesis with valuable ideas and advice, especially in the part devoted to the parsing of digital certificates, where he inspired most of the works. I am really pleased for all the notions that I learned from his "omniscience", formerly as a student and later as a colleague. I have also to thank him for the suggestion to become the teaching assistant for a course as complex as "Theoretical Computer Science and Algorithm Design", which was a useful and valuable activity, helping me a lot to improve my communication skills and giving me appealing human experiences with my students.

A stimulating environment for a PhD student is also crucial to produce good quality research and a memorable PhD experience. Luckily, I found myself in an astonishing environment, the Heap Lab, full of brilliant and funny people. Although I could not share many research activities with most of them, due to the heterogeneity of our group, being in such a friendly environment allowed me to really enjoy these years, without losing motivation even when the challenges of the research activity emerged. I would like to thank all the members of the Heap lab for all the moments full of "disagio" that we spent together, to which I hope I provided enough contributions. I have also to apologize with some of the whiteboards and chairs of the lab, as I was a bit bully with them in my raging moments.

I would like to also thank the reviewers of this thesis for their valuable comments who contributed to improve the quality of this manuscript, providing also some interesting insights for further developments of my research activity.

Last but not least, I must also mention my parents and my girlfriend, who supported me throughout all my PhD, bearing all the times I was unavailable during nights or weekends because of my research activities. Their support was extremely valuable especially during the harsh moments of my research activity.

Abstract

Cryptographic primitives are fundamental building blocks for the security of a system. Nonetheless, the security guarantees of cryptographic primitives do not depend only on their theoretical soundness, but also on their proper adoption in real-world applications. This process is often challenging, as there are many different issues that may compromise the security guarantees of cryptographic components, such as implementation flaws, side-channel attacks and misuses of the primitive. In this work, we focus on two relevant challenges that arise in the adoption of two cryptographic primitives in real-world applications: the unpractical performance overhead exhibited by cryptographic solutions for *privacy-preserving outsourced computation*; the security vulnerabilities stemming from the improper parsing of digital certificates.

Privacy-preserving outsourced computation techniques allow to offload a computation to an untrusted server while retaining the confidentiality of the data involved in the computation. Fully Homomorphic Encryption (FHE) is one of the most suitable techniques to perform privacy-preserving outsourced computation, as it allows arbitrary computation directly on encrypted data; nonetheless, its adoption in real-world applications is currently hindered by its prohibitive performance overhead. In this work, we aim at evaluating several strategies to reduce such unpractical performance overhead exhibited by FHE schemes. First, we investigate the security guarantees of existing *noise-free* FHE schemes, which are appealing due to their higher efficiency than common noisy FHE schemes. Our investigations lead to the design of two novel attack techniques against FHE schemes, which amplify the impact of existing vulnerabilities in the target FHE scheme by relying on its homomorphic capabilities. Our attack completely breaks two existing noise-free FHE schemes, hence showing the difficulty of designing a secure noise-free FHE scheme. Given the unavailability of secure and efficient noise-free FHE schemes, we evaluate two alternative strategies for privacy-preserving outsourced computation: employing Partial Homomorphic Encryption (PHE) schemes, which restrict the set of computations that can be performed on ciphertexts to gain some efficiency over FHE ones; relying on trusted hardware, such as the Intel Software Guard Extension (SGX) technology. We show the effectiveness of these two approaches by applying them to the design of two Privacy-Preserving Substring Search (PPSS) protocols, which allow to outsource the look-up of strings in outsourced documents while retaining the search and access pattern privacy of the queries as well as the confidentiality of the outsourced documents. Both our solutions show an extremely low bandwidth and a practical response time on real-world use cases, highlighting the effectiveness of our approach in reducing the performance overhead for privacy-preserving outsourced computation.

Digital certificates are widely employed in secure communication protocols to ensure the authenticity of the binding between a public key and its owner. Despite their crucial role, existing parsers for these certificates still exhibits a significant lack of accuracy, which has already been exploited to conduct several powerful attacks against the protocols relying on these certificates. For X.509 digital certificates, automatically generating a parser from a grammar specification has already turned out to be really effective in improving the parsing accuracy; nonetheless, the formal grammar for X.509 digital certificates is extremely complex, and thus hardly usable in real-world implementations. To overcome this issue, in this work we propose a novel format for X.509 digital certificates; our format, while retaining the same expressiveness of existing X.509 certificates, can be described by a simple regular grammar, in turn allowing the automatic generation of a parser exhibiting optimal time and space complexities. In addition, given the accuracy showed by automatically generated parsers for X.509 digital certificates, in this work we also analyze the format of certificates and messages in the OpenPGP protocol, showing that the OpenPGP format can be described by a Deterministic Context-Free grammar, from which an efficient parser can be automatically derived. Nonetheless, we show that such grammar requires a prohibitively high number of rules and it is thus unusable in practice. Furthermore, we outline several attacks that rely on different flaws that we identify in our analysis of the OpenPGP format, assessing their effectiveness against existing implementations of the OpenPGP protocol.

Estratto in Italiano

Le primitive crittografiche sono dei componenti fondamentali per la sicurezza di un sistema informatico. Tuttavia, le garanzie di sicurezza di queste primitive non dipendono esclusivamente dalla loro validità formale, ma anche dalla corretta integrazione delle stesse nelle applicazioni pratiche. L'adozione di questi componenti è spesso complicato, a causa delle molte insidie che possono compromettere le loro garanzie di sicurezza, come ad esempio degli errori implementativi, gli attacchi side channel e un improprio utilizzo della primitiva. In questa tesi, consideriamo due problemi rilevanti che emergono nell'adozione in applicazioni reali di due importanti primitive crittografiche: le notevoli perdite di performance causate dall'utilizzo di primitive crittografiche che consentono di delegare la computazione a una macchina non sicura senza compromettere la confidenzialità dei dati coinvolti nella computazione; le vulnerabilità di sicurezza derivanti da un'impropria validazione della struttura sintattica dei certificati digitali.

La crittografia omomorfa permette di effettuare una computazione direttamente su dati cifrati, consentendo quindi di delegare la computazione a una macchina non sicura mantenendo la confidenzialità dei dati coinvolti nella computazione. Purtroppo una grossa limitazione all'utilizzo pratico di questa primitiva è la significativa perdita di performance della computazione, che diventano proibitive rispetto allo svolgimento dello stesso calcolo su dati non cifrati. In questa tesi, consideriamo diverse possibili strategie per risolvere questa importante limitazione della crittografia omomorfa. In primo luogo, valutiamo le garanzie di sicurezza fornite da un tipo di schemi di crittografia omomorfa caratterizzati dall'assenza di rumore, che sono di significativo interesse a causa della loro efficienza rispetto a schemi più comuni. In particolare, introduciamo due nuove tecniche di attacco contro schemi di crittografia omomorfa, sfruttando le loro capacità omomorfe per amplificare l'impatto di alcune vulnerabilità esistenti, portando così alla completa compromissione delle garanzie di confidenzialità degli schemi vulnerabili ai nostri attacchi. Le nostre tecniche sono efficaci contro due schemi di crittografia omomorfa senza rumore, gli unici di questa tipologia disponibili con garanzie di sicurezza tali da renderli utilizzabili in applicazioni reali, mostrando le difficoltà nel progettare schemi di crittografia omomorfa senza rumore in maniera sicura.

Data l'indisponibilità di schemi di crittografia omomorfa senza rumore che coniugano sicurezza ed efficienza, consideriamo in questa tesi due strategie alternative per

consentire di delegare la computazione a una macchina non sicura mantenendo la confidenzialità dei dati coinvolti senza sacrificare l'efficienza del calcolo. Il primo approccio utilizza i cosiddetti schemi parzialmente omomorfi, che consentono di effettuare un insieme ristretto di calcoli su dati cifrati rispetto agli schemi di crittografia omomorfa, ma ne migliorano notevolmente le performance; il secondo approccio basa le sue garanzie di sicurezza su componenti hardware, come la tecnologia SGX di Intel, che consentono di eseguire una computazione in maniera sicura su una macchina il cui software, incluso quello di sistema, è potenzialmente compromesso da un attaccante. In questa tesi, mostriamo l'efficacia di questi due approcci progettando due protocolli che consentono di delegare la ricerca di sottostringhe in un insieme di documenti memorizzato in una macchina non sicura, garantendo la confidenzialità sia dei documenti che della sottostringa cercata, e senza rivelare le similarità tra le diverse ricerche. Il primo protocollo ha un ridotto consumo di banda e consente di effettuare simultaneamente ricerche diverse da parte di più utenti, rappresentando anche la prima soluzione che supporta la ricerca di stringhe contenenti metacaratteri (pattern matching) senza diminuire le garanzie di confidenzialità dei dati; il secondo protocollo ha un consumo di banda ottimale e offre anche garanzie di correttezza del risultato della ricerca, ma attualmente non supporta né ricerche simultanee da parte di più utenti, né le ricerche di stringhe contenenti metacaratteri. In conclusione, entrambi i protocolli mostrano un consumo di banda molto ridotto e dei tempi di risposta per le ricerche che evidenziano la loro applicabilità in casi d'uso reali, sottolineando così l'efficacia di entrambi gli approcci nel garantire la confidenzialità dei dati coinvolti in una computazione effettuata su una macchina remota senza introdurre delle eccessive perdite di performance.

I certificati digitali sono largamente utilizzati nei protocolli di comunicazione sicuri al fine di garantire la corrispondenza tra una chiave pubblica e il suo proprietario. Nonostante il ruolo fondamentale di questi certificati nel prevenire attacchi di impersonificazione, i riconoscitori sintattici di questi certificati mostrano ancora una notevole mancanza di accuratezza, che è già stata sfruttata per effettuare alcuni attacchi estremamente efficaci contro diversi protocolli la cui sicurezza è basata sui certificati digitali. Per i certificati X.509, utilizzati nel diffusissimo protocollo di comunicazione sicura TLS, la generazione automatica di un riconoscitore sintattico a partire da una grammatica formale ha dimostrato la sua efficacia nel migliorare l'accuratezza del riconoscimento sintattico; tuttavia, la grammatica formale utilizzata per i certificati X.509 è estremamente complessa, e di conseguenza difficilmente utilizzabile e manutenibile in applicazioni reali. Per risolvere questo problema, in questa tesi proponiamo un nuovo formato per i certificati digitali X.509. Il nostro formato, pur mantenendo la stessa capacità espressiva del formato attuale, può essere descritto con una semplice grammatica regolare, consentendo così la generazione automatica di un riconoscitore sintattico con complessità computazionali minime sia in termini di tempo che di memoria. Il nostro formato ha un preambolo che consente alle implementazioni esistenti di identificarlo immediatamente come una versione ignota, evitando in questa maniera di causare errori imprevedibili nelle implementazioni esistenti dovuti al tentativo di processare un formato sconosciuto; questa possibilità è cruciale nel consentire una graduale adozione del nostro formato nell'infrastruttura dei certificati digitali, in quanto consente la coesistenza tra implementazioni esistenti in grado di riconoscere solo i certificati attuali e i nuovi certificati costruiti con il nostro formato.

Considerando i buoni risultati in termini di accuratezza mostrati nei certificati X.509 dai riconoscitori sintattici automaticamente generati a partire da una grammatica, in questa tesi compiamo anche un primo passo in questa direzione per OpenPGP, il protocollo distribuito che rappresenta la principale alternativa al sistema centralizzato di autenticazione delle chiavi pubbliche basato sui certificati X.509. In particolare, presentiamo un'analisi del formato dei certificati e dei messaggi utilizzato nel protocollo OpenPGP, al fine di determinare se è possibile generare automaticamente un riconoscitore sintattico per questo formato. La nostra analisi mostra che il formato di OpenPGP può essere descritto da una grammatica libera dal contesto deterministica, da cui è possibile generare automaticamente un riconoscitore sintattico. Tuttavia, la nostra analisi rivela anche che la grammatica che descrive il formato ha un numero di produzioni estremamente elevato, che rendono impraticabile il suo utilizzo in qualsiasi infrastruttura di calcolo attualmente esistente. Di conseguenza, discutiamo in questa tesi possibili modifiche al formato di OpenPGP che consentono di ottenere una semplice grammatica libera dal contesto deterministica da cui è praticabile generare automaticamente un'automa a pila deterministico in grado di riconoscere certificati e messaggi di OpenPGP. In aggiunta, presentiamo diversi attacchi basati su alcune falle identificate durante la nostra analisi del formato del protocollo OpenPGP, valutando l'efficacia di questi attacchi contro le implementazioni esistenti del protocollo OpenPGP. La valutazione sperimentale mostra che i nostri attacchi non sono applicabili su queste implementazioni, principalmente a causa della bontà delle scelte degli sviluppatori, che hanno adottato la specifica OpenPGP considerando le implicazioni delle loro scelte sulla sicurezza dell'implementazione. Tuttavia, questo non dimostra che i nostri attacchi siano innocui contro una generica implementazione; per questo motivo, proponiamo anche delle modifiche alla specifica del formato di OpenPGP che consentono di evitare gli attacchi identificati.

Contents

List of Figures

List of Tables

Introduction

The design of secure systems often hinges upon the security guarantees provided by cryptographic primitives, which are among the most critical components of a secure system. Indeed, weaknesses found in cryptographic components are often sufficient to completely subvert the security guarantees of a system. Unfortunately, the adoption of theoretically sound cryptographic constructions does not necessarily guarantee that the cryptographic components of a system cannot be exploited by attackers. Indeed, both the design and the security guarantees of cryptographic primitives usually rely on complex statistical and algebraic concepts; because of such complexity, their implementation is rather error-prone, and even a simple flaw is often sufficient to completely subvert their security guarantees. A well-known example is the private key recovery attack against flawed implementations of Ellyptic Curve Digital Signature Algorithm (ECDSA) signature scheme. To compute the signature, this algorithm employs a value, called *nonce*, which must be sampled uniformly at random, as the signing private key can be efficiently computed if two signatures computed with the same nonce are collected by the attacker. This requirement of ECDSA was neglected by developers of Sony PlayStation 3, which naively employed the same nonce for every signature [1]; the same vulnerability was exploited against an Android Bitcoin wallet which computed ECDSA signatures with a nonce generated by a flawed random number generator [2].

Even if a cryptographic algorithm is properly implemented with no statistical or algebraic flaws, there is a further threat that the implementation has to withstand, namely the information leaked to the attacker through *side-channels*. In general, a side-channel is an information leak that derives from the execution of the algorithm on a device, such as the execution time or the power consumption. When this information may be actually exploited by an attacker to learn sensitive data (e.g., the cryptographic key), the implementation is said to be vulnerable to a side-channel attack. A well-known example of side-channel vulnerability is found in the square-and-multiply algorithm to compute exponentiation, which is a common operation in several asymmetric encryption schemes, such as RSA. This algorithm iterates over each bit of the exponent, performing a costly multiplication if and only if the bit at hand is 1. Thus, if an attacker

[1]https://fahrplan.events.ccc.de/congress/2010/Fahrplan/events/4087.en.html

[2]https://bitcoin.org/en/alert/2013-08-11-android

is able to profile the execution time of each iteration, it can easily learn the value of each bit of the exponent, which must be kept secret in several encryption algorithms: for instance, the decryption operation in the RSA scheme is an exponentiation where the exponent is the secret key; if the square-and-multiply algorithm is employed for this exponentiation, then the attacker can easily reconstruct the RSA key. To avoid this threat, it is necessary to employ a modified algorithm that performs the multiplication for each bit of the exponent, independently from their value.

In addition, even assuming that a safe implementation resilient to side-channel attacks is employed, application developers can easily misuse the cryptographic primitive. For instance, even by considering a simple cryptographic component such as a block cipher (e.g., Advanced Encryption Standard (AES)), an application developer with little knowledge of cryptography may compromise the confidentiality of the encrypted content by employing a secret key derived from a user password without adding further randomness to the password, or by choosing Initialization Vectors (IVs) of ciphertexts in increasing order, hence making them trivially predictable.

Finally, the adoption of cryptographic constructions is sometimes hindered by the computational or communication overhead they introduce, which is generally referred to as *security cost*. Indeed, application designers are willing to rely on the security guarantees of a cryptographic construction for their products only if the cryptographic construction does not significantly worsen the performance of the application; otherwise, it is possible that the cryptographic construction is discarded by the application designers, actually leaving the application with weaker or even no security guarantees.

In conclusion, besides the necessary theoretical soundness of a cryptographic component, there are several further challenges that need to be addressed to make the cryptographic component amenable to a secure and effective adoption in real-world applications. This general and complex problem requires specific solutions for each cryptographic primitive, as the implementation difficulties and performance bottlenecks usually vary between different primitives. The common thread of this thesis deals with tackling some of the aforementioned challenges for two specific cryptographic primitives, in order to foster their effective adoption in real-world applications. In particular, we focus on the security cost currently introduced to outsource a computation in a privacy-preserving fashion, especially when Homomorphic Encryption (HE) is employed, and on the inaccuracy of existing implementations in the processing of digital certificates. In the remainder of this chapter, we introduce the notions of privacy-preserving outsourced computation and digital certificate, we describe the challenges related to each of these cryptographic primitives that are tackled in this work, and we discuss our contributions to the solution of the described issues.

Privacy-Preserving Outsourced Computation

Outsourcing computation and data to entities with powerful computation and storage capabilities, such as data centers or cloud providers, is an emerging trend as it enables significant cost savings with respect to the deployment and the maintenance of an high end computing infrastructure. Nonetheless, offloading computation to a cloud provider requires to outsource also the data involved in the computation, in turn compromising the confidentiality of such data. Indeed, even if the data are outsourced in encrypted form, the cloud provider must be able to decrypt the data in order to perform the computation, which means that the data is necessarily disclosed to the cloud provider as well as being possibly leaked throughout security breaches targeting the premises of the cloud provider. This privacy loss is sometimes sufficient to discourage an entity from outsourcing computation to the cloud, especially if the outsourced data are particularly sensitive, as it is the case for biomedical or financial data.

To address the confidentiality issues of the outsourced data, we need to employ privacy-preserving computation techniques, which allow a set of parties to jointly perform a computation with each party learning no more than the information that can be inferred from its input values and the output of the computation. The outsourcing scenario can be conceived as a 2-party privacy-preserving computation, where the outsourcing entity inputs its data and receives the output of the computation while the cloud provider provides no input data (being it entirely provided by the outsourcing entity) and receives no output value. Thus, privacy-preserving techniques may foster the outsource of a computation even in applications processing particularly sensitive data, as they ensure that the cloud provider learns nothing from the outsourced computation, in turn effectively removing any concern about data confidentiality.

Among the variety of techniques for privacy-preserving computation, Homomorphic Encryption (HE) is perfectly suitable for the outsourcing scenario, as it allows to perform computation over encrypted data with neither decrypting the data nor learning the secret key. Indeed, the outsourcing entity can simply encrypt the input data with an HE scheme and send it to the cloud provider, which can compute the result from the input ciphertexts without the need to know the secret key; then, the encrypted result is sent back to the outsourcing entity, which recovers it by decrypting the received ciphertext. The cloud provider learns nothing throughout the computation, as it processes only ciphertexts; similarly, the data cannot be leaked through security breaches at the premises of the cloud provider, since the secret key to decrypt the ciphertext is never sent to the cloud provider by the outsourcing entity.

An HE scheme, besides the two usual encryption and decryption routines, is enriched with the homomorphic evaluation operation, called EVAL, which, given some input ciphertexts and a function f to be evaluated, computes a ciphertext that encrypts the evaluation of the function over the corresponding plaintexts of the input ciphertexts. The function f is generally represented by an arithmetic circuit with addition and multiplication gates over a given ring (e.g., the set of integers $\mathbb{Z}$). This representation is employed because the homomorphic evaluation of a function in an HE scheme hinges upon two homomorphic operations ADD and MUL, which homomorphically evaluate the addition and multiplication gates, respectively, over the ciphertexts fed as inputs to these operations; the EVAL operation is then performed by applying in place of each gate found in the arithmetic circuit the corresponding homomorphic operation.

HE schemes are classified according to the set of functions that can be homomorphically evaluated over encrypted data, which is referred to as the *homomorphic capability* of the scheme at hand. In particular, Partial Homomorphic Encryption (PHE) schemes can evaluate circuits with a single type of gate, i.e., they provide either ADD or MUL operation but not both of them; SomeWhat Homomorphic Encryption (SWHE) schemes can evaluate circuits with both addition and multiplication gates, but limited depth; lastly, Fully Homomorphic Encryption (FHE) schemes can evaluate arbitrary circuits, with no limitation on their depth.

PHE schemes are in general traditional cryptosystems where there is either an additive or a multiplicative homomorphism between the plaintext and ciphertext spaces. For instance, in the textbook RSA scheme, the encryption function is the exponentiation over $\mathbb{Z}_N$ (with N the composite RSA modulus) with a fixed exponent e, which is a multiplicative automorphism: indeed, given the public key e and two plaintexts $m_1, m_2 \in \mathbb{Z}_N$ with corresponding ciphertexts $c_1 = m_1^e \bmod N$, $c_2 = m_2^e \bmod N$, respectively, the ciphertext $c = c_1 \cdot c_2 \bmod N = m_1^e \cdot m_2^e \bmod N = (m_1 \cdot m_2)^e \bmod N$ is an encryption of $m_1 \cdot m_2 \bmod N$. In this case, the homomorphic multiplication MUL is as simple as a single multiplication over $\mathbb{Z}_N$, where N is a composite integer represented by few Kibit.

In order to get a FHE scheme, it is sufficient that the mapping between plaintexts and ciphertexts is both an additive and a multiplicative homomorphism. The most common strategy adopted to employ such an homomorphic mapping in a secure fashion (i.e, guaranteeing that plaintexts cannot be recovered from corresponding ciphertexts without the knowledge of the secret key) is performing a preliminary encoding of the plaintexts before applying the homomorphic mapping; nonetheless such encoding preserves the homomorphic properties of the mapping between plaintexts and ciphertexts only for a limited number of homomorphic operations. Such encoding is referred to as *noisy*, since the behavior obtained by introducing it is similar to the ones of systems dealing with noise. Indeed, the role of the encoding can be conceived as hiding the plaintext value by introducing a certain amount of noise, which can be removed up to a certain threshold. The noises of ciphertexts is combined and amplified by homomorphic operations, especially multiplication, in turn making it eventually too high to be removed. In this case, the plaintext value can no longer be recovered and the HE scheme exhibits a decryption failure; thus, this strategy allows to design only SWHE schemes, as the number of operations is limited by the noise growth. In order to reduce the impact of noise on the homomorphic capabilities, SWHE schemes are generally equipped with additional procedures that reduce the noise growth after homomorphic operations, such as modulus switching [22] or scale invariant techniques [20, 55]. These procedures, called *noise management* techniques, introduce a significant performance overhead to homomorphic computation in SWHE schemes. A further source of computational overhead in SWHE schemes is the ciphertext expansion, which may be as high as 2 or 3 order of magnitudes. Indeed, the ciphertexts are generally complex algebraic structures, and thus the homomorphic operations, which are necessarily performed over elements of these structures, become also much more complex and computationally intensive than the corresponding operations on the plaintext values.

The limitation on the number of operations that can be homomorphically performed in a SWHE scheme can be removed by applying the *bootstrapping* procedure, proposed by Gentry in 2009. This was a significant breakthrough in the design of HE schemes, as it lead to the construction of the first FHE scheme [60], which had been an open problem since 1978, when Rivest [129] introduced the concept of FHE with the name of *privacy homomorphism*. Differently from other noise management techniques that simply limit the noise growth, bootstrapping allows to obtain ciphertexts with a fixed amount of noise without decrypting the ciphertexts. When bootstrapping is applicable to a SWHE scheme, it become possible to evaluate an arbitrary circuit, since the noise can always be reduced to a fixed amount that allows to perform further homomorphic operations. Nonetheless, bootstrapping is an even more complex operation than other noise management techniques, hereby introducing a much higher computational overhead. For this reason, FHE schemes with a slower noise growth are generally preferable despite the availability of the bootstrapping procedure, because they allow to bootstrap less frequently, in turn amortizing the bootstrapping cost over more homomorphic operations.

In conclusion, we observe that the performance overhead of HE schemes increase with their homomorphic capabilities, mainly because the homomorphic evaluation become much more complex. In particular, the performance overhead exhibited by FHE has prevented a wider adoption of these schemes in real-world applications, as the security cost is still deemed too high. Since the introduction of the bootstrapping technique by Gentry, several FHE schemes were proposed [154, 15, 21, 63, 84, 35, 99, 54, 101], based on different security assumptions [126, 144, 8] and yielding remarkable performance improvements. In addition, researchers enriched also the features for privacy-preserving computation available in FHE schemes: the *batching* technique [61, 141] allows to construct ciphertexts with multiple slots, each storing its own plaintext value, and to apply an homomorphic operation simultaneously to all the slots in a Single Instruction Multiple Data (SIMD) fashion; *multi-key* FHE schemes [99, 113, 100] allow to perform computation over ciphertexts encrypted with independent keys from different owners, and the result of the computation can be decrypted by a secret key that is jointly computed by all the owners of the independent keys; lastly, the CKKS scheme [34] allows to perform homomorphic evaluation of ciphertexts representing rational values. We refer interested readers to [108, 2] for a detailed outline of the most relevant HE schemes proposed in the literature as well as the efforts aimed at achieving efficient and optimized implementations of such schemes. Despite all these significant improvements, the security cost of FHE is still quite unpractical: for instance, TFHE library [36], one of the most efficient existing implementations of a FHE scheme, reports about tens of milliseconds to execute one homomorphic operation, corresponding to approximately 6 order of magnitudes overhead w.r.t. computation over plaintext values. In this thesis, we are interested in investigating possible solutions to overcome this high security cost while allowing to outsource a computation in a privacy-preserving manner.

Noise-Free Schemes

Given the performance overhead introduced by noise management techniques and bootstrapping, an interesting solution to reduce such overhead is represented by *noise-free* FHE schemes. As suggested by their name, these schemes do not apply a noisy encoding to the plaintext values: they simply employ an additive and multiplicative homomorphism between the plaintext and ciphertext space, which allows to perform an arbitrary number of operations without incurring in a decryption failure. Being noise-free, these schemes do not require costly noise management techniques and thus they represent a viable strategy to reduce the computational overhead exhibited by FHE schemes. Nonetheless, most of the noise-free FHE schemes found in the literature exhibit critical security weaknesses, which allow to easily recover the plaintext values or even the secret key from ciphertexts. Indeed, the noise-free scheme proposed in [98] was subsequently showed to be broken in [158], while the scheme found in [85] can be completely subverted by knowing the corresponding plaintext value of only two ciphertexts [151]. Two noise-free schemes that have not showed any security weakness yet are the ones found in [94, 115]: nonetheless, both of them employ cumbersome algebraic structures as their plaintext spaces (i.e., non commutative rings and group presentations), and finding a mapping between integer values and elements of such structures that preserves both the homomorphic properties and the security guarantees of the schemes is still an open challenge; since this mapping is necessary to compute ciphertexts for the input data involved in the outsourced computation, these schemes are currently unusable in real-world applications. To the best of our knowledge, the only existing noise-free FHE schemes that may be employed in real-world applications are `OctoM` [158] and `JordanM` [158], which are based on octonion and Jordan algebras, respectively. Since these schemes are *linearly decryptable*, i.e., their decryption function can be expressed as the inner product between the ciphertext and the secret key, they exhibit two security weaknesses: they are vulnerable to Known Plainext Attacks (KPAs), that is the attacker can recover the secret key by knowing the corresponding plaintext value for slightly less than a hundred of ciphertexts; there exists an efficient *distinguisher* algorithm that allows an attacker with no knowledge of the secret key to determine if the corresponding plaintext value of a generic ciphertext is equal to the integer value 1 or not. Despite these vulnerabilities, which are acknowledged by the designers of the schemes too, `OctoM` and `JordanM` are claimed to be secure in a ciphertext-only scenario, unless it is computationally feasible to solve quadratic modular equations over a ring $\mathbb{Z}_N$, with N being a composite integer, a problem which is as hard as factoring N.

`OctoM` and `JordanM` are appealing solutions for privacy-preserving computation from a performance standpoint, as the homomorphic ADD and MUL operations are as simple as matrix addition and multiplication, respectively, with no noise management techniques being involved. Nonetheless, to foster their usage, there is the need to better understand if they can be securely employed despite the vulnerabilities acknowledged by their authors. Indeed, the existing vulnerabilities may not prevent the usage of the scheme in application scenarios where it is unlikely that the attacker may learn a sufficient number of ciphertexts with known plaintext values to mount a KPA, or where it is extremely unlikely to have ciphertexts whose corresponding plaintext is the integer 1. Unfortunately, our analysis, which is reported in this manuscript, shows that the existence of a distinguisher for a single plaintext value for both these schemes make them insecure even in a scenario where the attacker has no additional information on the processed ciphertexts (i.e., a ciphertext-only scenario). In particular, we show that the existence of a distinguisher for a single plaintext value in a FHE scheme is sufficient to mount a plaintext recovery attack for any ciphertext. We deem our attack as *comparison-based*, since the core technique relies in combining the distinguisher and the homomorphic evaluation of the comparison between two integer values in order to efficiently guess the plaintext value of a generic ciphertext. The main drawback of our attack resides in its computational complexity: indeed, our attack tests increasing possible plaintext values for a given ciphertext until it finds the correct one, thus yielding a computational effort proportional to the plaintext value being recovered. Although this limitation allows to successfully perform our attack only against ciphertexts with relatively small plaintext values (e.g., no bigger than 2^{32}), we observe that in FHE schemes the magnitude of plaintext values is not expected to be much higher, as they correspond to the input values of the computation. We remark that if the adversary knows a range of possible plaintext values for a given ciphertext, then our attack can test only values found in this range, yielding a computational complexity which is proportional to the size of the range at hand rather than the actual plaintext value. Furthermore, in the experimental evaluation of our attack against `OctoM` and `JordanM`, we show that our attack is massively parallel, that is the speed-up of a parallel implementation scales with the number of processing nodes available. In addition, in the specific case of `OctoM` and `JordanM`, we show that we can employ our attack to recover the plaintext value of a sufficient number of ciphertexts to directly recover the secret key through KPA. To address these security issues, we also present a simple modification to `OctoM` and `JordanM` that mitigates our attack; nonetheless, the schemes remain vulnerable to a KPA with the same number of known plaintext values. Unfortunately, with our second attack technique we also show that the existence of a KPA is sufficient to completely break a FHE scheme even with a limited amount of information on the plaintext value of a single ciphertext, in turn preventing the secure adoption of the modified `OctoM` and `JordanM` to make a computation privacy-preserving.

Privacy-Preserving Outsourced Computation with PHE Schemes

Given the unavailability of efficient and secure FHE noise-free schemes, we explore a different strategy to make privacy-preserving outsourced computation more practical: hinging upon weaker but more efficient HE schemes. Indeed, as we discussed earlier, the performance overhead of HE schemes generally increase with their homomorphic capabilities. In particular, if we consider PHE schemes, their performance overhead w.r.t. plaintext computation is much smaller than FHE ones, given the simplicity of homomorphic operations and the absence of noise management techniques. The challenge in employing these schemes in place of FHE ones resides in devising an efficient strategy to perform the computation with the single homomorphic operation available in the PHE scheme being employed. In order to achieve the best performance, this design process must be necessarily tailored to the requirements of each privacy-preserving outsourced computation. In this thesis, we tackle this challenge for an important building block of privacy-preserving applications, that is the computation of substring search queries based on an inverted index. In particular, we consider the scenario reported in Fig. 1. A data owner with a set $\mathbf{D}$ of documents, called document collection, computes an inverted index that allows to efficiently compute the positions of all the repetitions, or *occurrences*, of a string over the document collection $\mathbf{D}$. The inverted index, referred to as *full-text index* from now on, is outsourced in encrypted form to an untrusted server (e.g., a cloud provider). The data owner, and possibly other users authorized by the data owner, can then send queries for a string q; the untrusted server employs the full-text index to compute the result of the query, namely the positions of all the occurrences of the string q in the document collection, and sends it back to the user. Our goal is achieving a practical response time for the queries while guaranteeing that the computation is privacy-preserving, which means that the untrusted server learns no more information than the size n of the document collection $\mathbf{D}$, the length m of the string q and the number o_q of its occurrences over $\mathbf{D}$. Since each user is assumed to run on a constrained device with limited storage and computational capabilities, we set as

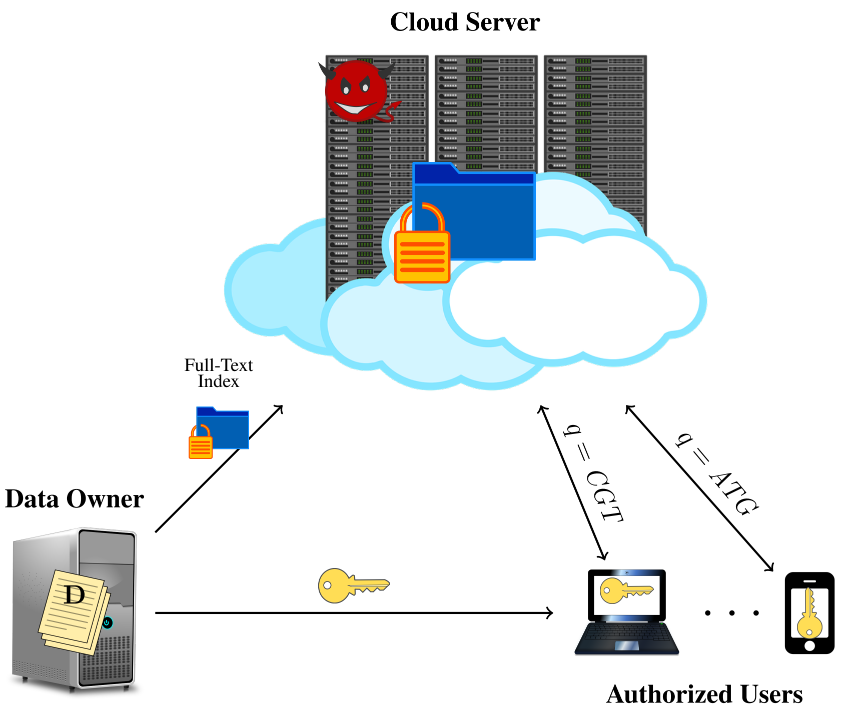


Figure 1: *Entities involved in a PPSS protocol for a document collection* $\mathbf{D}$

a necessary requirement for our solution a lightweight computation for the user issuing the query; thus, the performance metrics that mostly affect the response time are the computational cost at server side and the amount of information exchanged between the user and the server during a query, referred to as communication cost. Specifically, in the design of our Privacy-Preserving Substring Search (PPSS) protocol, we aim at reducing as much as possible the communication cost rather than the computational one, as long as the response time remains acceptable for the end user. Indeed, while the query is executed by a cloud service provider, which can in general provide enough computational power to meet the Service Level Agreement (SLA) requirements, the end user issuing a query may reside on a device with a low latency connection and a non flat-rate plan for network access (e.g., a mobile phone); in such scenario, a protocol with low bandwidth and communication cost is more appealing than a protocol with a lower response time but exhibiting a prohibitively high bandwidth consumption.

There exists different solutions to tackle the problem of securely outsourcing substring search queries, which exhibit different trade-offs between performance and privacy guarantees. The first class of solutions [92, 147, 33, 69], referred to as *substring-searchable symmetric encryption* schemes, allows to perform queries on an untrusted server with both computational and communication costs that depend only on the length m of the queried string and on the number o_q of its occurrences in the document collection (thus independent from the size n of the collection); nonetheless, all these solutions leak to the untrusted server both the *search* and the *access patterns*: the former refers to any information regarding the similarities between queries (e.g., if the same string is searched more than once), while the latter refers to any information related to the results of the query (e.g., if some queries share a common set of occurrences). The leakage of such information, when paired with public domain knowledge about the documents of the collection, was proven [30, 67, 130] to be sufficient to recover a significant portion of the documents or the content of the queried string, in turn pushing for the adoption of solutions preventing such information leakage. This need lead to the design of several PPSS protocols [155, 80, 139, 112, 56, 123] that hide search or access patterns from the untrusted server. Nonetheless, the reduced information leakage comes at the cost of making communication and computational costs no longer independent from the size n of the document collection.

To improve over the practicality of existing PPSS solutions that protect search and access patterns, we propose the first PPSS protocol with $\mathrm{O}(m)$ communication rounds and $\mathrm{O}(m\log^2(n)+o_q\log(n))$ overall communication cost that enables multiple simultaneous queries from different users and guarantees both search and access patterns privacy against a *semi-honest* adversary who follows the protocol specification but is eager to learn as much information as possible about the data involved in the computation. The computational cost of our PPSS protocol at server side amounts to $\mathrm{O}(mn)$, i.e., it is independent from the number of retrieved occurrences o_q, with $\mathrm{O}(n)$ storage cost; our protocol allows authorized users to issue the queries from constrained devices, as our protocol is computationally light at client side, exhibiting $\mathrm{O}((m+o_q)\log^4(n))$ computational cost and $\mathrm{O}(\log(n))$ memory. Remarkably, our protocol does not require any interaction between the users and the data owner throughout the queries, and its memory consumption at server side scales well with the number of simultaneous queries being performed, as it requires only $\mathrm{O}(\log^2(n))$ additional memory per query instead of replicating the whole outsourced full-text index for each query. To the best of our knowledge, our PPSS protocol is the first one that allows to perform queries in a privacy-preserving manner for strings containing wildcard characters; furthermore, it provides a lightweight mechanism to verify the correctness of the data retrieved by the server, which makes our protocol resilient against accidental or misconfiguration errors. The experimental validation on a real-world genomic use case of our PPSS protocol highlights its practical performance for substring search queries: a parallel implementation of our solution allows to execute a query over the 21^{st} human chromosome, which contains about 40 MiB of data, in few minutes, requiring only 50 KiB of bandwidth per communication round.

Privacy-Preserving Outsourced Computation with Intel SGX

In the light of our goal of reducing the performance overhead of privacy-preserving outsourced computation, in this work we also evaluate the usage of trusted hardware in place of relying only on cryptographic primitives. Specifically, we address the design of privacy-preserving applications based on the trusted execution environment, referred to as *secure enclave*, provided by Intel Software Guard Extension (SGX) technology [42], which have been available in Intel CPUs since the *Skylake* microarchitecture. Secure enclaves are protected memory regions that store code and data belonging to the application that instantiates the enclave. The SGX technology enforces at hardware level that only the application that instantiates the enclave can access the code and data stored in the enclave, thus ensuring their confidentiality and integrity even against the Operating System (OS) or the hypervisor of the machine hosting the enclave. Indeed, the content of an enclave is always stored in encrypted form with a key known only to the CPU and decrypted only when moved inside the CPU. To ensure the confidentiality and integrity of the code and data found in the enclave, SGX enforces a strong isolation between the code running in the enclave and any external component. In particular, the SGX programming model forbids both system calls and calls to shared libraries from code run inside the enclave, as these strict constraints prevent an implicit and subtle information leak between trusted and untrusted code. Conversely, the enclave can communicate only with the untrusted portion of the application that instantiates the enclave, which corresponds to the code of such application running outside the enclave; the SGX model allows to define an interface between the the trusted and the untrusted portions of the application, ensuring that these two portions exchange data only through the methods of this interface. Since the untrusted portion of the application resides outside the enclave, it can be manipulated by the attacker, hence SGX enforces strong isolation of the trusted portion by flushing all the CPU registers, which stores plaintext data, when the execution context switches between trusted and untrusted modes, and vice versa.

Thanks to these security guarantees, SGX technology enables the execution of an application on an untrusted machine while guaranteeing the confidentiality and integrity of the data employed by the application, which is useful for instance in digital right management scenarios to protect copyrighted material from being disclosed without authorization by legitimate consumers of the content. Since the enclave must be necessarily instantiated by the untrusted machine where the enclave resides, the security guarantees of SGX may be easily circumvented if the malicious OS of the hosting machine instantiates an enclave with a modified code that may send the data in unencrypted form outside the enclave; to avoid such threat, SGX provides a *remote attestation* procedure, which allows the remote owner of the application to verify that the untrusted machine has correctly instantiated the enclave with the expected code and data. Specifically, the remote attestation produces a fingerprint of the enclave signed with a key known only to the CPU, and whose authenticity can be automatically verified only by Intel through the Intel Attestation Service (IAS); in addition, the remote attestation allows to establish a secret session key between the enclave and the application owner, which is used to avoid Man in the Middle (MitM) attacks by the untrusted machine aimed at learning the information exchanged between the enclave and the application owner.

Securely outsourcing a computation to a cloud provider with SGX is rather simple. Indeed, the outsourcing entity asks the cloud server to instantiate an SGX enclave with its own application, which must be properly modified to be run inside an SGX enclave, and verifies through the remote attestation the correct setup of the enclave. Then, it employs the secret session key to send in encrypted form the data needed for the computation, which is safely stored inside the enclave. At the end of this setup phase, the application running inside the enclave starts the computation over the received data; when the results are available, the outsourcing entity retrieves them through the secure channel established with the enclave. Throughout this process, the outsourcing entity has to trust only the hardware vendor (i.e., Intel) that guarantees the authenticity of the remote attestation. The big advantage of hinging upon SGX for privacy-preserving outsourced computation is the modest performance overhead w.r.t. pure cryptographic solutions: indeed, after the data is decrypted when moved in the CPU, the computation proceeds over unencrypted data with no actual overhead.

Nonetheless, the main weakness of SGX resides in the multiple side channel attacks that have been proposed since its deployment in Intel CPUs, which may pose some concerns about the security of this technology. We can split these attacks in *microarchitectural* and memory access pattern reconstruction ones. The former leverages the side effects on the state of the CPU (e.g., cache or internal buffers) caused by microarchitectural features of the CPU such as speculative or out-of-order execution, employing techniques that follow the blueprint of the seminal works on the Spectre [90] and Meltdown [97] attacks. In particular, specific microarchitectural attacks targeting SGX technology have been proposed in the last three years [27, 26, 124]: these attacks completely subvert the security guarantees of SGX, as they allow the adversary to read the whole enclave memory and recover the per-CPU keys that ensure the authenticity of the attestation procedure. The main concerns about the security of SGX technology stem from the fact that new attacks have been found quite regularly, which generally allow to circumvent the mitigations to preceding attacks promptly released by Intel mostly as microcode updates [75, 76, 74]. Furthermore, each of these countermeasures generally introduce a non negligible performance overhead (which can be as high as $20\times$ [76]) on the CPU, and thus stacking all those mitigations may lead to a significant performance degradation. That being said, all the microarchitectural attacks proposed so far were mitigated by Intel, thus SGX technology can be currently considered secure.

Memory access pattern reconstruction attacks are weaker than microarchitectural ones, as they do not recover the per-CPU keys but they only allow to reconstruct the access pattern to code and data of the portion of the application running inside the enclave. Nonetheless, as the branches and memory accesses of an application are usually data dependent, this information can be sufficient to infer a significant portion of the data stored in the enclave. For instance, in [161], authors managed to recover a large portion of a JPG image from the memory access pattern of the compression algorithm run inside a secure enclave. Two side channels are mostly employed by existing attacks to learn the memory access pattern of the application run inside the enclave: the sequence of page faults experienced by the application [161], which can be easily observed by a privileged adversary running on the untrusted machine as SGX hands over the management of virtual memory and paging to the OS; the latency experienced by an application controlled by the attacker on cache lines shared with the victim application running inside the enclave [25, 23]. Differently from microarchitectural attacks, Intel explicitly claimed to exclude these attacks from its threat model [41], which means that there is no intention to mitigate them.

This statement fostered several research efforts aimed at preventing or detecting these side channel attacks. In [140], a compiler-level mitigation that makes page faults independent from the control flow of the application was proposed, while T-SGX [138] employs Transactional Synchronization Extension (TSX) technology to conceal page faults events to the OS. While these countermeasures exclusively target page fault based attacks, Cloak [68] refines T-SGX by protecting also against the cache attacks presented in [23]; nonetheless, it is not effective against the ones described in [25]. Finally, Varys [116] ensures that no concurrent applications are run on the same core of the enclave application, protecting against page fault based attacks and all the attacks that leverage resources shared by the enclave with applications running on the same core (e.g., L1 caches); nonetheless, it is still vulnerable to attacks leveraging contention on the level 3 cache. In conclusion, none of these countermeasures can prevent all the existing side channel attacks.

Instead of preventing the leakage of the memory access pattern of the application running inside the enclave to the adversary, an alternative strategy is making such leakage meaningless. This goal can be achieved by making the application *oblivious*, which means that it exhibits the same memory access pattern independently from the data being processed. The existing approaches [4, 135, 111] achieve oblivious execution in SGX applications by hinging upon the Oblivious RAM (ORAM) primitive [64], which allows a client with limited storage capabilities to remotely access data outsourced to an untrusted server without revealing to the server, which observes only the physical accesses to the ORAM data structure, the access pattern to the outsourced data. The performance overhead introduced by ORAM is mostly due to the network latency between the client and the server, as the client needs to fetch more data than needed in order to conceal which element is retrieved. However, when an ORAM is employed in SGX applications, the client can be safely moved inside the enclave, as the confidentiality guarantees of SGX allows to conceal the client data structures from the untrusted machine; in this way, the latency to access data stored in the ORAM can be significantly reduced. Nonetheless, as leaking the access patterns of the ORAM client algorithms through SGX side channels is sufficient to invalidate the privacy guarantees of ORAM, it is necessary to make the ORAM client oblivious too. Coherently with the terminology employed in [111], henceforth an ORAM with an oblivious client is referred to as Doubly Oblivious RAM (DORAM). ZeroTrace [135] was the first work that proposed the adoption of a DORAM to ensure that an application running inside an SGX enclave performs oblivious accesses to its data structures. Specifically, ZeroTrace provides a DORAM-based memory controller in a standalone enclave, which can be employed by a generic application to store and obliviously access its data through specific APIs. Nonetheless, ZeroTrace does not protect the memory access pattern to code pages, which may still reveal sensitive data. This limitation was overcome by Obfuscuro [4], which, given the source code of a generic application, instruments and compile the code in order to execute the application obliviously inside an SGX enclave, employing two DORAMs to protect accesses to code and data pages, respectively. Nonetheless, Obfuscuro introduces a non negligible performance overhead, since it has to perform a costly access to both the DORAMs every 3 to 5 executed assembly instructions.

In our vision, the solution achieving a reasonable trade-off between performance and security guarantees of privacy-preserving applications based on SGX is an hybrid approach between ZeroTrace and Obfuscuro: hinging upon a DORAM to conceal the accesses to sensitive data and employing an algorithm with a data-independent control flow to protect the accesses to code pages, hence making the application oblivious. This solution has already been adopted in the design of Oblix [111], a search index that enables efficient privacy-preserving keyword based look-ups inside an SGX enclave. In this thesis, we apply this design principle to obtain the first PPSS protocol based on SGX, with the aim of evaluating the benefits and the drawbacks of this technology w.r.t. other PPSS solutions relying only on cryptographic primitives. Our PPSS protocol, called Oblivious Substring Queries on Remote Enclave (ObSQRE), achieves an **optimal** communication cost of $\mathrm{O}(m+o_q)$ in one communication round, since the client simply sends the string q to the enclave and receives back the occurrences at the end of the query, while exhibiting a remarkably $\mathrm{O}((m+o_q)\log^3(n))$ computational cost at server side. Furthermore, by protecting the memory access pattern of the application running inside the enclave, ObSQRE conceals both the search and the access patterns of the substring search queries, even against a powerful *malicious* adversary who may tamper with the protocol execution in order to affect the correctness of the computation or learning as much information as possible. We remark that the ObSQRE threat model encompasses only attacks excluded from SGX threat model, as we rely on Intel to mitigate attacks that completely subvert SGX security guarantees. Unfortunately, ObSQRE currently exhibits less features than our previous PPSS protocol based on PHE, as ObSQRE does not support multiple simultaneous queries from different users and it does not allow to perform queries for strings containing wildcard characters. Differently from existing DORAM based solutions that protect SGX based applications from side-channels [4, 135, 111], which all employ a doubly-oblivious design of Path [142] ORAM, in ObSQRE we evaluate, with the aim of finding the best performing DORAM for our PPSS protocol, three different doubly-oblivious designs of three existing ORAMs: specifically, we propose a new design of Path DORAM, which improve upon existing ones employed in all the previous works, and we propose the first doubly oblivious designs of Circuit [156] and Ring ORAMs [128]. These DORAMs may be of independent interest w.r.t. ObSQRE, as they can be employed as a building block for other privacy-preserving SGX applications or for DORAM based countermeasures against SGX side channel attacks.

Digital Certificates

Digital certificates are the mainstay to establish a secure communication channel among different entities, as they ensure the authenticity of the binding between public keys and the entities owning them, effectively thwarting MitM attacks. In particular, the authenticity of the binding is guaranteed by digital signatures included in the certificate. The entities entitled to compute signatures on digital certificates vary depending on the trust model employed to ensure the authenticity of public keys. In the hierarchical and centralized approach of Public Key Infrastructure (PKI), there is a set of specific trusted entities called Certification Authorities (CAs), whose main purpose is verifying the owner of a public key and issuing a digitally signed certificate that attests the ownership of the public key; each end entity deems as valid a digital certificate if and only if it is signed by a trusted CA. PKI is mostly employed in the ubiquitous Transport Layer Security (TLS) protocol. Conversely, in the the peer-to-peer and distributed approach of Open Pretty Good Privacy (OpenPGP), each entity can act both as a CA and and as an end-user: it can digitally sign the public key of another entity after personally verifying the ownership of the public key, and it decides which signatures can be deemed as trusted. Specifically, OpenPGP is based on the idea that trust can be seen as a transitive relation: each entity may personally verify the authenticity of the keys of a restricted set of other entities, and then recursively extend its trust to all the keys that are signed by a key which is already trusted. This network of trust relationships, called Web of Trust (WoT), allows an entity to assess the authenticity of public keys whose ownership cannot be directly verified. OpenPGP is mostly employed for end-to-end email encryption/authentication and to ensure the integrity of software downloaded from repositories in several Linux distributions.

Despite the simplicity of their functionality, digital certificates contain much more information than the public key and the identity of its owner, such as:

- Information on how to verify the digital signatures on certificates, including the identify of the entity that signed the certificate
- Information on the validity of the public key, which may be revoked or expired
- Information on which usages are allowed for the public key (e.g., digital signatures, encryption, key agreement).

All these information are provided in a structured format, which needs to be properly parsed in order to correctly process such information. The format is obviously different between digital certificates employed in PKI and OpenPGP: the former employs the X.509 format, standardized by International Telecommunication Union (ITU) [77] and by Internet Engineering Task Force (IETF) in Request For Comment (RFC) 5280 [39] and its complements [72, 122, 93, 152, 45]; the latter employs its own format, standardized by IETF in RFC 4880 [29]. Both formats share a significant complexity, which makes the proper parsing of digital certificates a non trivial task. We now outline the issues and the security vulnerabilities emerged from the inaccurate processing of digital certificates, and our contributions towards the goal of a sound and effective parsing for digital certificates.

X.509 Format

The complexity of the X.509 format, which has now reached its third version, motivates the existence of several parsing inconsistencies found in widely employed libraries. Many of these inconsistencies turned out to be actually exploitable to completely subvert the security guarantees provided by X.509 digital certificates, effectively leading to powerful attacks. In particular, several well-known parsing libraries were found to be vulnerable to some MitM attacks, which relied on either inconsistencies among different libraries in the processing of the information found in an X.509 digital certificate about the owner of the public key [106, 107], or vulnerabilities (e.g., integer overflows) in the processing of specific fields of such certificates [81]. Improper parsing issues lead also to other type of attacks, such as certificate forgery: in the *BERSERK* attack [47, 46], authors show that an attacker can inject arbitrary content in a certificate and forge a valid signature for the crafted certificate, exploiting an improper counting of the number of bytes stored in a specific field of the certificate.

Besides single attempts at finding specific flaws or vulnerabilities in the parsing of digital certificates, there are also existing works aimed at find parsing inconsistencies in an automatized and systematic fashion. Specifically, [24] automatically generated crafted and invalid X.509 digital certificates, called *Frankencerts*, to assess the effectiveness of TLS libraries in validating X.509 certificates and to identify parsing inconsistencies among different implementations. Frankencerts were constructed by merging different portions of existing X.509 digital certificates in such a way that the generated certificate was a well-formed X.509 one (i.e., it exhibits all the major fields of an X.509 digital certificate) but was likely invalid as it violated some constraints of the format, in turn allowing to identify more specific parsing inconsistencies than the ones obtained by randomly generating a digital certificate entirely from scratch. Frankencerts were employed to perform differential testing on most common TLS implementations and allowed to discover many dangerous flaws that may easily enable MitM attacks. A finer-grain validation of X.509 parsing effectiveness was achieved by the *RFCcert* tool [149]. The approach followed in this work was deemed as RFC guided, as the tool automatically extracted a set of constraints from the RFC specification of the format and then generated a set of invalid certificates, each violating one of the identified constraints. This test set allowed to assess the effectiveness of X.509 parsers in dealing with specific syntactic constraints, which is extremely helpful in the precise identification of the flaw and in its resolution. The validation of 14 TLS implementations with the generated test set allowed to identify 29 inconsistencies on average per implementation, with some of them being exploitable as security issues and leading to powerful attacks, such as certificate forgery.

All these recent efforts showed that parsing inconsistencies have been continuously found for X.509 digital certificates, despite their format has been well-known and widely used for more than 30 years. This fact suggests that a proper and effective parsing of X.509 digital certificates has not been achieved yet. A weakness of existing X.509 parsers that motivates their inaccuracy resides in the lack of a systematic approach employed in their construction, as parsers found in most common TLS libraries are all handcrafted. Conversely, as mandated by *language-theoretic* security principles [134], a reliable strategy to properly parse an input format is hinging upon a parser automatically generated from a grammar specification of the format. Indeed, writing a formal grammar for a given input format is much less error prone than coding entirely from scratch a parser for the format, as formal grammars are aptly conceived to easily describe the syntactic constraints found in a format specification. In case of X.509 digital certificates, the lack of an automatically generated parser is mainly motivated by the complexity of the format: indeed, if we consider the well-known Chomsky's classification of formal languages [37], the set of valid X.509 digital certificates is a *context-sensitive* language [81], which hinders the automatic generation of a parser as there is currently no strategy to automatically derive an efficient parser from a context-sensitive grammar. This issue was addressed in [13], where authors managed to design a grammar amenable to the automatic generation of an efficient parser for the X.509 format. In this work, authors designed a *predicated* grammar [120] to deal with most of the syntactic constraints in the format, enriching some of the rules with additional pieces of code, referred to as *semantic actions*, which were in charge of performing context-sensitive checks. Once obtaining a sound and complete predicated grammar, enriched with semantic actions, for X.509 format, an efficient parser for X.509 digital certificates was derived through the ANTLR parser generator [119]. The comparison of this parser with 7 common TLS libraries showed the higher accuracy of the automatically generated parser w.r.t. handcrafted implementations, as tens of parsing inconsistencies in the TLS libraries were identified. The exploitation of one of these inconsistencies lead to a powerful MitM attack against the widely employed OpenSSL [117] and BoringSSL [65] TLS libraries. Furthermore, the automatically generated parser found out that about 21% of the 11M X.509 digital certificates employed for secure HTTPS connections over the entire IP v4 space were syntactically invalid w.r.t. the X.509 standard.

Although [13] represents an important step towards the proper parsing of X.509 digital certificates, the significant complexity of the predicated grammar for X.509 format makes the development and maintenance of this solution in real world implementations quite complicated. Therefore, in this work we aim at addressing the root cause of all the parsing difficulties described so far, that is the complexity of the X.509 format. To this extent, we design a novel format for X.509 digital certificates, which may represent a good candidate as a possible fourth version of the X.509 standard. Our new format retains the same expressiveness of the current version of X.509 standard, but it can be specified by a **regular** grammar rather than a context-sensitive one. This feature significantly simplifies the grammar describing the format and allows to automatically generate a parser with optimal computational complexities, as parsing a string belonging to a regular language can be performed in linear time and constant space complexities. Furthermore, we design our format in order to ensure that existing parsers identify certificates in our format as version 4 ones: this solution eases the roll-in of our format, as implementations that are not yet compliant to our format can immediately stop processing our certificates, because of a wrong version field found at the beginning of the X.509 digital certificate, instead of exhibiting an unexpected behavior due to the validation of a malformed input.

OpenPGP format

Differently from X.509 format, the one of OpenPGP digital certificates and messages has received poor attention, mostly because of the wider adoption of PKI over OpenPGP. This discrepancy is likely due to the fact that digital certificates in PKI can be entirely handled automatically by software without requiring any human intervention, which is impossible in OpenPGP as trust is based on the keys that can be personally verified by each user. Nonetheless, since OpenPGP is employed in critical scenarios, such as guaranteeing the authenticity of the software fetched from repositories in many Linux distributions, it is important to deepen the scrutiny also on OpenPGP implementations. In this regard, the few existing works are mostly focused on the state of the WoT ecosystem. Specifically, [91] analyzed the email addresses that generally identify users in the WoT ecosystem, finding out that about 40% of such addresses were unreachable. [19] focused on the analysis of the strength of cryptographic keys found in OpenPGP certificates, identifying several weak key pairs. Authors of [10] showed how weaknesses in the cryptographic keys may affect much many certificates than the ones belonging to the owners of the weak keys, because of the transitive trust model employed in OpenPGP and to the strong interconnections between users in the WoT. Specifically, authors showed that compromising the few weak keys found in the *strong set*, which represents the most interconnected users in the WoT, was sufficient to make any signature forged by the attacker with the compromised key pairs trusted for about 70% of such strong set, and thus to all the users trusting at least one of these members of the strong set.

Although these work highlighted important issues for the reliability of the WoT, none of them focused on the assessment of the accuracy of existing OpenPGP implementations in processing digital certificates and messages. Nonetheless, as showed by the numerous attacks caused by an improper parsing of X.509 digital certificates, it is crucial to ensure the availability of sound implementations that prevent security threats caused by an inaccurate processing of the OpenPGP digital certificates and messages. The strategy to achieve this goal is similar to the one that has already proven to be successful for X.509 digital certificates, that is automatically generating a parser from a grammar specification of the format. To this extent, in this manuscript we perform a detailed analysis of the OpenPGP format, investigating the feasibility of designing a grammar amenable to automatic parser generation algorithms. Our analysis shows that the OpenPGP format can be specified by a Deterministic Context-Free (DCF) grammar; nonetheless, we also show that the Deterministic PushDown Automaton (DPDA) derived from this grammar has so many states that it cannot even be represented in nowadays computing devices without exceeding the available memory. Therefore, the DCF grammar for OpenPGP format cannot be employed in practice to automatically generate a parser, thus pushing for alternative strategies to systematically derive a parser for the OpenPGP format. Throughout our analysis, we also identify several design flaws in the OpenPGP format, conceiving several attacks that stem from the exploitation of such flaws. We evaluate the effectiveness of our attacks on the two most common implementations of OpenPGP protocol, namely GNU Privacy Guard (GPG) [89], the open source one employed in Unix based systems, and the proprietary implementation Symantec PGP [148].

Organization of the Manuscript

We split the thesis in two parts, the former one dealing with privacy-preserving outsourced computation, and the latter dealing with digital certificates. In each of these parts, we start with a chapter where we provide the background information and the definitions which are needed to describe our contributions. Then, we devote a single chapter to each work being discussed in this thesis. Specifically, Chapter 2 describes our comparison-based attack techniques against noise-free FHE schemes; Chapter 3 describes our multi-user PPSS protocol based on PHE; Chapter 4 describes ObSQRE, our PPSS protocol based on Intel SGX; Chapter 6 describes our novel format for X.509 digital certificates; lastly, Chapter 7 reports our security audit of the OpenPGP format. After these two parts, we conclude the manuscript with a final chapter, where we summarize the contributions and a discussion of the results of this research, also discussing possible further developments that stem from the conclusions derived from our findings.

List of Papers

Most of the contributions discussed in this manuscript have already been published in several peer-reviewed papers. Specifically:

- The description of our comparison-based attack, its experimental evaluation and the modifications to `OctoM` and `JordanM` that mitigate our attack are reported. in [103]. A preliminary version of this attack was presented in [12]. Our second attack technique, which allows to break the modified `OctoM` and `JordanM` scheme, is instead a novel contribution reported exclusively in this manuscript.

- The description of our multi-user PPSS protocol based on a PHE scheme is reported in [102]. Nonetheless, some of the features reported in this manuscript, such as the privacy-preserving queries for strings containing wildcard characters or the integrity check mechanism, are described in an extended version of the work which is currently under review for the ACM international journal on Digital Threats: Research and Practice (DTRAP).

- The description of ObSQRE, our PPSS protocol based on Intel SGX, is reported in [104]. A partial but substantial contribution to the design and the implementation of ObSQRE was also given in [132].
- The description of our novel regular format for X.509 digital certificates is reported in [14].
- Our security audit of the OpenPGP format is reported in [11].

Part I

Privacy-Preserving Outsourced Computation

CHAPTER 1

Definitions and Preliminaries

In this chapter, we introduce all the definitions and preliminary concepts that are needed to describe our research activity aimed at reducing the performance overhead of privacy-preserving outsourced computation. In particular, we organize this chapter as follows:

- In Section 1.1, we provide a formal definition of FHE schemes followed by the definition of Length-Flexible Additive Homomorphic Encryption (LFAHE) scheme, the type of PHE scheme which will be employed in one of our PPSS protocols.
- In Section 1.2, we describe existing solutions to homomorphically evaluate the comparison function between two integer values, which is a fundamental building block in our comparison-based attack against noise-free FHE schemes.
- In Section 1.3, we introduce the notation and we formally define the notion of PPSS protocol, also describing in detail the existing solutions found in the literature.
- In Section 1.4, we describe the *backward-search* algorithm, which employs a full-text index built from the Burrows-Wheeler Transform (BWT) and the Suffix Array (SA) of a document in order to efficiently perform substring search queries over the document. The substring search algorithms employed in our two PPSS protocols are all based on the backward-search method
- In Section 1.5, we introduce the notion of Private Information Retrieval (PIR) and we describe the Lipmaa's protocol, which is combined with the backward-search algorithm in order to build our multi-user PPSS protocol.
- Lastly, in Section 1.6, we describe the three existing ORAMs employed in ObSQRE, our PPSS protocol based on Intel SGX.

We report here our definition of negligible function, which is extensively used in this part of the manuscript.

Definition 1.1 (Negligible Function). *A function $\epsilon : \mathbb{N} \rightarrow \mathbb{R}$ is negligible if, for every univariate positive polynomial, $poly(x) \in \mathbb{R}[x]$, there exists an integer $c > 0$ such that $\forall\, x > c$, $|\epsilon(x)| \leq \frac{1}{poly(x)}$.*

1.1 HE Schemes

We start with our definition of a generic HE scheme, and then we introduce the concepts of Additive Homomorphic Encryption (AHE) and LFAHE schemes.

An HE scheme specifies three sets: $\mathcal{M}$, $\mathcal{C}$ and $\mathcal{F}$. The set of plaintexts $\mathcal{M}$ is usually a ring algebraic structure; in this work, we consider a plaintext space given by a set of integer values ranging from 0 to $N{-}1$, with $N{\geq}2$, which are assumed to be the representatives of the residue classes modulo N, i.e., $(\mathbb{Z}_N, +, \cdot)$, where $\mathbb{Z}_N \equiv \mathbb{Z}/N\mathbb{Z}$. The ciphertext space $\mathcal{C}$ includes elements with an algebraic representation that depends on the specific HE scheme at hand. The set of multivariate polynomials $\mathcal{F} \subseteq \mathbb{Z}_N[x_1, x_2, \ldots, x_a]$, with $a \geq 1$ and degree greater or equal to zero, defines the functions that the HE scheme at hand allows to be evaluated. Each of these polynomials computes a function $f : \mathcal{M}^a \rightarrow \mathcal{M}, a \geq 1$, over the plaintexts, which is represented by an *arithmetic circuit* composed by gates performing modular multiplications $(\cdot)$ and modular additions $(+)$ in $\mathbb{Z}_N$.

We provide the definition of an HE scheme starting from an asymmetric one, and then we describe a symmetric HE scheme by difference.

Definition 1.2 (Public-key Homomorphic Encryption Scheme). *A public-key HE scheme is defined as a tuple of four polynomial time algorithms $\langle$KEYGEN, ENC, DEC, EVAL$\rangle$:*

- ***Key Generation**. $(sk, pk, evk) \leftarrow$ KEYGEN(1^λ) is a probabilistic algorithm that, given the security parameter λ, generates the secret key sk, the public key pk and the public evaluation key evk.*
- ***Encryption**. $c \leftarrow$ ENC(pk, m) is a probabilistic algorithm that, given a message $m \in \mathcal{M}$ and the public key pk, computes a ciphertext $c \in \mathcal{C}$.*
- ***Decryption**. $m \leftarrow$ DEC(sk, c) is a deterministic algorithm that, given a ciphertext $c \in \mathcal{C}$ and the secret key sk, outputs a message $m \in \mathcal{M}$.*
- ***Evaluation**. $c \leftarrow$ EVAL$(evk, f, c_1, c_2, \ldots, c_a)$ is a probabilistic algorithm that, given an arithmetic circuit $f \in \mathcal{F}$ with $a \geq 1$ inputs, the ciphertexts $c_1, c_2, \ldots, c_a$, and the evaluation key evk, computes a ciphertext $c \in \mathcal{C}$.*

The following properties must hold:

- ***Decryption Correctness**. $\forall\, m \in \mathcal{M}$: DEC$\,(sk,$ ENC$(pk, m)) = m$.*
- ***Evaluation Correctness**. $\forall m_1, \ldots, m_a \in \mathcal{M}, f \in \mathcal{F}$:*
  $\Pr($DEC$(sk,$ EVAL$(evk, f, c_1, \ldots, c_a)){=}f(m_1, \ldots, m_a)) = 1 - \epsilon(\lambda)$,
  where $c_1{=}$ENC$(pk, m_1), \ldots, c_a{=}ENC(pk, m_a)$ and $\epsilon(\lambda)$ is a negligible function in the security parameter.

- ***Compactness**. $\forall f \in \mathcal{F}$, $c_1, \ldots, c_a \in \mathcal{C}$:*
  $|\textsc{Eval}(evk, f, c_1, \ldots, c_a)| \leq poly(\lambda)$, where $|\cdot|$ denotes the bit-length of a ciphertext, while $poly(\cdot)$ denotes a univariate polynomial.

When defining a symmetric-key homomorphic encryption scheme, the only difference is the key generation algorithm $\textsc{KeyGen}(1^\lambda)$ outputting a tuple $\mathtt{k} = (sk, pk, evk)$ with $sk = pk$.

Algorithm 1.1: Efficient algorithm to compute HybridMul operation based on the square-and-multiply algorithm

```
Input: Ciphertext $c \in \mathcal{C}$
       Integer $h \geq 1 \in \mathbb{Z}$
       Evaluation key $evk$
Output: Ciphertext $c' = \textsc{HybridMul}(c, h)$
begin
    res ← ⊥, mask ← 1, tmp ← $c$
    while $mask \leq h$ do
        if $h$ & mask ≠ 0                /* & denotes the bitwise and operation */
         then
            if res = ⊥ then
                res ← tmp
            else
                res ← $\textsc{Add}(evk$, res, tmp$)$
        tmp ← $\textsc{Add}(evk$, tmp, tmp$)$
        mask ← 2 · mask
    return res
```

The requirement on the evaluation correctness trivially states that by decrypting the output of the Eval algorithm we obtain the evaluation of the function f over the corresponding plaintext values of the a ciphertexts $c_1, \ldots, c_a$. In particular, the Eval algorithm homomorphically evaluates a function f by hinging upon the homomorphic operations Add and Mul, which are defined as follows:

- **Homomorphic Addition**. $c \leftarrow \textsc{Add}(evk, c_1, c_2)$ is a probabilistic algorithm that, given the evaluation key evk and the ciphertexts $c_1, c_2 \in \mathcal{C}$, computes a ciphertext $c \in \mathcal{C}$ such that $\textsc{Dec}(sk, c) = \textsc{Dec}(sk, c_1) + \textsc{Dec}(sk, c_2)$.

- **Homomorphic Multiplication**. $c \leftarrow \textsc{Mul}(evk, c_1, c_2)$ is a probabilistic algorithm that, given the evaluation key evk and the ciphertexts $c_1, c_2 \in \mathcal{C}$, computes a ciphertext $c \in \mathcal{C}$ such that $\textsc{Dec}(sk, c) = \textsc{Dec}(sk, c_1) \cdot \textsc{Dec}(sk, c_2)$.

By repeatedly applying each of these two operations, it is possible to perform the following operations:

- **Hybrid Mul**. $c' \leftarrow \textsc{HybridMul}(evk, c, h)$ is a probabilistic algorithm that, given the evaluation key evk, a ciphertext $c \in \mathcal{C}$ and an integer h, computes a ciphertext $c' \in \mathcal{C}$ such that $\textsc{Dec}(sk, c') = \textsc{Dec}(sk, c) \cdot h$

- **Hybrid Exp**. $c' \leftarrow \textsc{HybridExp}(evk, c, h)$ is a probabilistic algorithm that, given the evaluation key evk, a ciphertext $c \in \mathcal{C}$ and an integer h, computes a ciphertext $c' \in \mathcal{C}$ such that $\textsc{Dec}(sk, c') = \textsc{Dec}(sk, c)^h$

In particular, HYBRIDMUL (resp. HYBRIDEXP) can be computed with $\mathrm{O}(\log(h))$ ADD (resp. MUL) operations by relying on Alg. 1.1, which is a modified version of the well-known *square-and-multiply* algorithm.

By employing these four operations, the EVAL function can homomorphically evaluate multivariate polynomials. The HE schemes are classified according to the set $\mathcal{F}$ of polynomials that can be evaluated. In particular, an AHE scheme can evaluate only linear polynomials, as only ADD and HYBRIDMUL are available; a Multiplicative Homomorphic Encryption (MHE) can evaluate only monomials, as only MUL and HYBRIDEXP are available; in a SWHE scheme, the degree of the polynomials that can be evaluated is bounded by an integer $d \geq 2$; lastly, a FHE scheme can evaluate arbitrary polynomials.

Length-Flexible Additive Homomorphic Encryption Schemes

We now introduce the definition of LFAHE scheme and we describe the LFAHE scheme by Damgård and Jurik [44], which is employed in the construction of our multi-user PPSS protocol.

Definition 1.3 (Length-Flexible Additive Homomorphic Encryption)**.** *A LFAHE scheme is an AHE scheme augmented with an additional parameter $l \geq 1$, called length, which specializes the definition of the plaintext and ciphertext spaces, and thus of the encryption, decryption and homomorphic addition operations, such that:*

$$\forall l_1, l_2 \in \mathbb{N}(l_1 < l_2 \Rightarrow \mathcal{C}^{l_1} \subset \mathcal{M}^{l_2})$$

where the superscript l_1 (resp. l_2) is employed to specify the plaintext and ciphertext spaces for length l_1 (resp. l_2).

Specifically, the expression $\mathcal{C}^{l_1} \subset \mathcal{M}^{l_2}$ indicates that ciphertexts in $\mathcal{C}^{l_1}$ are valid plaintexts for ciphertexts in $\mathcal{C}^{l_2}$ (i.e., a ciphertext in $\mathcal{C}^{l_1}$ is a valid output of the decryption algorithm fed with an element of $\mathcal{C}^{l_2}$).

Damgård-Jurik LFAHE Scheme. The Damgård-Jurik (DJ) LFAHE scheme [44] is constructed from the AHE scheme proposed by Paillier [118]. The Paillier cryptosystem is a public key AHE scheme based on the *Composite Residuosity Class Problem*, which is reducible in polynomial time to the *Integer Factorization Problem* [118]. The Paillier scheme is semantically secure, which intuitively means that it is computationally unfeasible to determine if two ciphertexts encrypt the same plaintext or not. The plaintext space of this scheme is $\mathcal{M} = \mathbb{Z}_N$, with N being an integer of few Kbits computed as the product of two large primes, while the ciphertext space is $\mathcal{C} = \mathbb{Z}^*_{N^2} \subset \mathbb{Z}_{N^2}$, i.e., the subset of all and only elements of $\mathbb{Z}_{N^2}$ with a multiplicative inverse modulo N^2. The key generation algorithm computes the public key pk and the private key sk, with the public evaluation key evk being the same as pk.

Given the ciphertexts $c_1, c_2 \in \mathbb{Z}^*_{N^2}$, the homomorphic addition is a ciphertext $c = c_1 \cdot c_2 \bmod N^2$, $c \in \mathbb{Z}^*_{N^2}$, such that $\textsc{Dec}(sk, c) = \textsc{Dec}(sk, c_1) + \textsc{Dec}(sk, c_2) \bmod N$. Therefore, the result of an hybrid homomorphic multiplication HYBRIDMUL is obtained as an exponentiation of a ciphertext c to an integer. It can also be conceived as the encryption of the product of two plaintext values in $\mathbb{Z}_N$:

$$\forall m \in \mathbb{Z}_N, c \in \mathbb{Z}^*_{N^2} : \textsc{Dec}(sk, c^m \bmod N^2) = \textsc{Dec}(sk, c) \cdot m \bmod N$$

By combining the homomorphic addition and the HYBRIDMUL operation, the Paillier scheme allows to perform the *dot product* between a cell-wise encrypted array $\langle A \rangle$ and an unencrypted one B, both with $n \geq 1$ elements; this operation, referred to as *hybrid dot product*, is computed as follows:

$$\textsc{Dec}\left(sk, \prod_{i=1}^{n} (\langle A \rangle[i])^{B[i]} \bmod N^2\right) = \sum_{i=1}^{n} A[i] \cdot B[i] \bmod N \tag{1.1}$$

In the DJ scheme, a length-flexible variant of the Paillier one, the plaintext and ciphertext spaces depend on the length l of the encryption. In particular, a ciphertext of length $l \geq 1$ is an integer in $\mathcal{C}^l = \mathbb{Z}^*_{N^{l+1}}$ encrypting a plaintext value belonging to $\mathcal{M}^l = \mathbb{Z}_{N^l}$. Therefore, the homomorphic addition between two ciphertext $c_1, c_2 \in \mathcal{C}^l$ is a ciphertext $c = c_1 \cdot c_2 \bmod N^{l+1}$, $c \in \mathcal{C}^l$, such that $\textsc{Dec}(sk, c) = \textsc{Dec}(sk, c_1) + \textsc{Dec}(sk, c_2) \bmod N^l$.

The hybrid homomorphic multiplication HYBRIDMUL between a ciphertext $c_2 \in \mathbb{Z}^*_{N^{l_2+1}}$ of length l_2 and a ciphertext $c_1 \in \mathbb{Z}^*_{N^{l_1+1}}$ of length l_1, with $l_1 < l_2$, computes a ciphertext of length l_2 encrypting the product between the corresponding plaintext value of c_2 (which is an integer in $\mathbb{Z}_{N^{l_2}}$) and c_1, being $\mathbb{Z}^*_{N^{l_1+1}} \subset \mathbb{Z}_{N^{l_2}}$. Indeed, $\forall c_1 \in \mathbb{Z}^*_{N^{l_1+1}}, c_2 \in \mathbb{Z}^*_{N^{l_2+1}}$:

$$\textsc{Dec}_{l_2}\left(sk, c_2^{c_1} \bmod N^{l_2+1}\right) = \textsc{Dec}_{l_2}(sk, c_2) \cdot c_1 \bmod N^{l_2}$$

where the subscript l_2 denotes that the decryption operation is performed for plaintext and ciphertext spaces $\mathcal{M}^{l_2}$ and $\mathcal{C}^{l_2}$. This property of the HYBRIDMUL operation in LFAHE schemes is extremely useful, as it allows to perform the *homomorphic dot product* between an array $\langle A \rangle_{l_2}$ encrypted cell-wise with length l_2, and an array $\langle B \rangle_{l_1}$ encrypted cell-wise with length l_1, both with $n \geq 1$ elements:

$$\textsc{Dec}_{l_2}\left(sk, \prod_{i=1}^{n} (\langle A \rangle_{l_2}[i])^{\langle B \rangle_{l_1}[i]} \bmod N^{l_2+1}\right) = \sum_{i=1}^{n} A[i] \cdot \langle B \rangle_{l_1}[i] \bmod N^{l_2} \tag{1.2}$$

This homomorphic operation is at the core of the Lipmaa's PIR protocol, which will employed in the construction of our multi-user PPSS protocol.

1.2 Homomorphic Comparison of Integers

One of the fundamental building blocks of our comparison based attack for FHE schemes is the homomorphic evaluation of the comparison function between two integers values. In this section, we formally define this functionality and we discuss existing methods proposed in the literature for its homomorphic evaluation.

The comparison function we need to homomorphically evaluate is the *greater-than* one, which, given two integers, determines if the former is greater or equal than the latter. For our needs, it is sufficient to consider this function over an interval of integers rather than over the entire $\mathbb{Z}$. More formally:

Definition 1.4 (Greater-than Function). *Given a positive integer b and an interval of integers $D_t = \{0, 1, \ldots, t-1\}$, with $t \geq 2$, the greater-than function $GT_{t,b} : D_t \times D_t \rightarrow$*

$\{b-1, b\}$ *is defined as:*

$$GT_{t,b}(x,y) = \begin{cases} b & \text{if } x \geq y, \\ b-1 & \text{otherwise} \end{cases}$$

To the extent of evaluating this function with an HE scheme, we need to find a polynomial $f_{\texttt{gt}} \in \mathcal{F} \subseteq \mathbb{Z}_N[x,y]$, such that $f_{\texttt{gt}}(x,y) \bmod N = GT_{t,b}(x,y)$, with $2 \leq t \leq N$, $1 \leq b < N$, and x, y being the representatives of residue classes modulo N, (i.e., $x, y \in \mathbb{Z}_N$) considered as integers less than t. Although such a polynomial can be easily found for specific values of N (e.g., $N = 2$), its construction is challenging for a generic $N > 2$. Çetin in [31] reports two methods to homomorphically compute the $GT_{t,b}(\cdot,\cdot)$ function that do not require interaction between the secret key owner and the party who performs the homomorphic evaluation. However, both of these methods are not applicable for our attack: indeed, the first one is applicable only if the modulus N is prime; the second method computes an approximation of $GT_{t,b}(\cdot,\cdot)$, while for our attack we need an exact computation of this function.
A more effective solution is proposed in [114]: the greater-than function is computed as $GT_{t,b}(x,y) = \mathit{SIGN}_{t,b}(x-y)$, where $\mathit{SIGN}_{t,b}(z)$ is a function defined over the interval $\overline{D_t} \subseteq \mathbb{Z} = \{-t+1, \ldots, 0, \ldots, t-1\}$ as:

$$\mathit{SIGN}_{t,b}(z) = \begin{cases} b & \text{if } z \geq 0, \\ b-1 & \text{otherwise} \end{cases}$$

The homomorphic evaluation of the function $\mathit{SIGN}_{t,b}(\cdot)$ requires a polynomial $f_{\texttt{sign}} \in \mathcal{F} \subseteq \mathbb{Z}_N[z]$ fulfilling $f_{\texttt{sign}}(z) \bmod N = \mathit{SIGN}_{t,b}(z)$, with $2 \leq t \leq \lceil \frac{N}{2} \rceil$, $1 \leq b < N$ and $z \in \overline{D_t}$. We remark that this method does not work if $N = 2$, as there is no integer t such that $2 \leq t \leq \lceil \frac{N}{2} \rceil$; nonetheless, this is not a limitation as the homomorphic computation of $GT_{t,b}(\cdot,\cdot)$ function is straightforward in case $N = 2$. In [114], the polynomial $f_{\texttt{sign}}$ is computed applying the Lagrange interpolation formula to $2t-1$ points having coordinates $(\, z,\, \mathit{SIGN}_{t,b}(z)\,)$, with $z \in \overline{D_t}$, and considering a prime modulus, i.e., $N = p$. Nonetheless, we now show that by introducing an additional constraint on the domain of the $GT_{t,b}(\cdot,\cdot)$ function, it is possible to employ this method also for a generic modulus $N > 2$.

Lemma 1.1. *Given two integers N, t, with $N > 2$ and $2 \leq t \leq \lceil \frac{N}{2} \rceil$, and a set $\overline{D_t} = \{-t+1, \ldots, 0, \ldots, t-1\}$, the polynomial $f(z) \in \mathbb{Z}_N[z]$, interpolating $2t-1$ points $(z, f(z))$ having the* z*-coordinate ranging over all values in $\overline{D_t}$, exists if $t \leq \frac{q}{2}$, where q is the smallest prime factor of N.*

Proof. Considering the set of $2t-1$ points $\{(z_1, y_1), \ldots, (z_{2t-1}, y_{2t-1})\}$ in $\mathbb{Z}_N \times \mathbb{Z}_N$, the interpolating polynomial $f \in \mathbb{Z}_N[z]$, with degree at most $2t-2$, can be computed by the Lagrange interpolation formula:

$$f(z) = \sum_{i=1}^{2t-1} y_i \prod_{j=1, j\neq i}^{2t-1} (z - z_j)(z_i - z_j)^{-1}$$

The existence of the multiplicative inverses (in $\mathbb{Z}_N$) required in this formula is ensured if all the values $z_i - z_j$ are co-prime with N. Assuming the z-coordinates to be mutually distinct and in $\overline{D_t}$, the constraint $t \leq \frac{q}{2}$ implies that $-q < -2t+2 \leq z_i - z_j \leq 2t-2 < q$. Since q is the smallest prime factor of N, then all the elements in $\mathbb{Z}_N \setminus \{0\} \cap \{-q+1, \ldots, q-1\}$ are co-prime with N; thus, since all the values $z_i - z_j \in \mathbb{Z}_N \setminus \{0\} \cap \{-q+1, \ldots, q-1\}$, then these values are co-prime with N, and thus invertible, allowing $f(z)$ to be interpolated by the Lagrange formula. $\square$

In conclusion, by Lagrange interpolation we can obtain a polynomial $f_{\texttt{sign}} \in \mathcal{F} \subseteq \mathbb{Z}_N[z]$ which computes the function $SIGN_{t,b}(z), \forall z \in \overline{D_t}$, and then a polynomial $f_{\texttt{gt}} \in \mathcal{F} \subseteq \mathbb{Z}_N[x,y]$, computing the function $GT_{t,b}(x,y)$, $\forall x, y \in D_t$, as $f_{\texttt{gt}}(x,y) = f_{\texttt{sign}}(x-y)$.

Since $f_{\texttt{gt}} \in \mathcal{F}$, it can be homomorphically evaluated by the EVAL algorithm of the HE scheme at hand, replacing addition and multiplications of the polynomial with the corresponding homomorphic operations (ADD and MUL), whose inputs are ciphertexts in $\mathcal{C}$. Henceforth, we denote the algorithm $\textsc{Eval}(evk, c_1, c_2, f_{\texttt{gt}})$ as $HGT_{t,b}(c_1, c_2)$; since $GT_{t,b}$ is defined over the interval D_t=$\{0, \ldots, t-1\}$, $t \leq \frac{q}{2}$, with q being the smallest prime factor of N, then c_1, $c_2 \in \mathcal{C}_t = \{c \in \mathcal{C} \text{ s.t. } \textsc{Dec}(sk, c) < t\}$ is a sufficient condition for $\textsc{Dec}(sk, HGT_{t,b}(c_1, c_2)) = GT_{t,b}(\textsc{Dec}(sk, c_1), \textsc{Dec}(sk, c_2))$.

The computational complexity required to interpolate $2t-1$ points by applying the Lagrange formula is $\mathrm{O}(t^2)$ operations in $\mathbb{Z}_N$, while the evaluation of the polynomial $f_{\texttt{sign}} \in \mathbb{Z}_N[z]$, whose degree is at most $2t-2$, has a computational complexity $\mathrm{O}(t)$. Therefore, the computational cost of the $HGT_{t,b}(\cdot,\cdot)$ algorithm, assuming that the polynomial $f_{\texttt{sign}}$ is precomputed, is $\mathrm{O}(t)$. We note that, while there are no current algorithms to compute $HGT_{t,b}(\cdot,\cdot)$ in less than $\mathrm{O}(t)$, research efforts driven by the usefulness of homomorphic comparison as a building block for privacy-preserving applications may lead to an improvement in this sense. Since our attack relies on the computation of $HGT_{t,b}(\cdot,\cdot)$ as an atomic component, such improvements will positively affect the efficiency of our attack.

1.3 PPSS Protocol

In this Section, we formally define the notion of PPSS protocol and we discuss existing solutions, comparing their efficiency and capabilities with our proposals. We start with a brief introduction of the notation employed in our definitions and to describe PPSS protocols throughout this manuscript.

Notation. Given a string s over an alphabet Σ, the function $\textsc{len}(s)$ denotes the number of characters in s. We use Σ^m to denote the set of strings over the alphabet Σ of length m. For a string $s \in \Sigma^m$, we denote as $s[i, \ldots, j]$, with $i \leq j$ and $i, j \in \{0, \ldots, m-1\}$, the portion of the string s that starts with character in position i and ends with the character in position j (i.e., for s=privacy, $s[2, \ldots, 4]$=iva).

Definition of PPSS Protocol

In the scenario considered in this work, a data owner has a set of $z \geq 1$ documents $\mathbf{D} = \{D_1, \ldots, D_z\}$, referred to as document collection, where each document $D_i \in \Sigma^{\textsc{len}(D_i)}$. We denote by n the size of the document collection, which is the sum of the lengths of its documents (i.e., $n = \sum_{i=1}^{z} \textsc{len}(D_i)$). From this set of documents, the data owner builds a full-text index that allows to efficiently perform substring search queries over the document collection.

Definition 1.5 (Substring Search Query)**.** *Consider a collection of* $z{\geq}1$ *documents* $\mathbf{D} = \{D_1, \dots, D_z\}$*, with each* $D_i \in \Sigma^{\text{LEN}(D_i)}$*,* $1{\leq}i{\leq}z$*, and a query string* $q{\in}\Sigma^m$*,* $m{\geq}1$*.*

A substring search query computes the occurrences of q *in each document of* $\mathbf{D}$*, that is the set* $O_{\mathbf{D},q} = \bigcup_{i=1}^{z} O_{D_i,q}$*, where*

$$O_{D_i,q}{=}\{\, j \mid 0{\leq}j{\leq}\text{LEN}(D_i){-}m \wedge q{=}D_i[j, \dots, j{+}m{-}1] \,\}$$

Because of its limited storage and computational capabilities, the data owner is willing to outsource both the document collection $\mathbf{D}$ and the computation of substring search queries to a cloud provider. Since both the content of the documents and the strings searched in the queries are sensitive data, the data owner wants to be sure that no meaningful information about these data is leaked to the cloud provider throughout the queries. In particular, the cloud provider should learn no more than the number z of documents in $\mathbf{D}$, the size n of the document collection, the number m of characters of the searched string q and the number o_q of occurrences of q in $\mathbf{D}$ (i.e., $o_q = |O_{\mathbf{D},q}|$). The main challenge to be addressed resides in guaranteeing this limited information leakage while retaining a practical response time for these privacy-preserving substring search queries. Furthermore, in a multi-user scenario, the data owner can authorize other users to perform privacy-preserving substring search queries over the document collection.

A PPSS protocol is a solution that allows to satisfy these needs of the data owner. To guarantee the confidentiality of the documents and of the searched string, a PPSS protocol employs *privacy-preserving representations* of these data. Each PPSS protocol relies on specific cryptographic primitives to build the privacy-preserving representations of the data processed by the protocol. In this manuscript, we denote a privacy-preserving representation of a datum by enclosing it in angular brackets (i.e., $\langle\mathbf{D}\rangle$ is the privacy-preserving representation of $\mathbf{D}$). We now provide our formal definition of PPSS protocol; in this definition, we refer to the data owner as the client and to the cloud service provider as the untrusted server.

Definition 1.6 (PPSS Protocol)**.** *Consider a document collection* $\mathbf{D}$ *with* $z \geq 1$ *documents* $\mathbf{D} = \{D_1, \dots, D_z\}$ *over an alphabet* Σ*. A PPSS protocol* $\mathcal{P}$ *is a pair of polynomial-time algorithms* $\mathcal{P} = (\text{SETUP}, \text{QUERY})$*:*
The setup procedure*:* $(\langle\mathbf{D}\rangle, aux_s) \leftarrow \text{SETUP}(\mathbf{D}, 1^\lambda)$*, is a probabilistic algorithm, run by the data owner, taking as input the security parameter* λ *and the document collection* $\mathbf{D}$*, and returning its privacy-preserving representation* $\langle\mathbf{D}\rangle$*, which is outsourced to the untrusted server in the cloud, an an auxiliary pieces of information* aux_s*, which is kept secret by the data owner, and shared with entities authorized to issue queries in case of a multi-user scenario.*
The query procedure*:* $R \leftarrow \text{QUERY}(q, aux_s, \langle\mathbf{D}\rangle)$*, is a deterministic algorithm run interactively by the server and the client issuing the query, which is either the data owner or any other entity authorized by the data owner in case of a multi-user scenario. The* QUERY *procedure, given the string* q *to be searched and the secret auxiliary information* aux_s*, both provided by the client, employs the privacy-preserving representation* $\langle\mathbf{D}\rangle$*, stored at server side, to compute the result of the query* $R = O_{\mathbf{D},q}$*, where* $O_{\mathbf{D},q}$ *is the set defined in Def. 1.5, which is obtained by the client.*

The QUERY *procedure iterates $w \geq 1$ rounds, where each round corresponds to the execution of the following three algorithms:*

- TRAPDOOR*:* $\langle q \rangle_j \leftarrow$ TRAPDOOR$(j, q, aux_s, res_1, \dots, res_{j-1})$*, is a probabilistic algorithm, run at client side, which employs aux_s and the results of previous rounds $\{res_1, \dots, res_{j-1}\}$ to build the privacy-preserving representation (a.k.a. trapdoor) $\langle q \rangle_j$ of the searched string q for the j-th round.*
- SEARCH*:* $\langle res_j \rangle \leftarrow$ SEARCH$(\langle q \rangle_j, \langle \mathbf{D} \rangle)$*, is a deterministic algorithm, run at server side, which employs $\langle q \rangle_j$ and $\langle \mathbf{D} \rangle$ to compute a privacy-preserving representation of the result for the j-th round, i.e., $\langle res_j \rangle$.*
- RETRIEVE*:* $res_j \leftarrow$ RETRIEVE$(\langle res_j \rangle, aux_s)$*, is a deterministic algorithm, run at client side, which, given as inputs $\langle res_j \rangle$ and aux_s, computes the result of the j-th round res_j.*

Existing Solutions

We now discuss the existing solutions that allows to perform privacy-preserving substring search queries, also comparing them with our proposals. In order to perform a fair comparison, we describe only protocols that conceal the search or access patterns of the queries, as solutions with weaker privacy guarantees employ simpler cryptographic primitives (such as symmetric encryption or order-preserving encryption schemes), in turn making them much more efficient than PPSS protocols guaranteeing search or access pattern privacy.

In [155], the authors proposed a PPSS protocol to establish if a given substring is present in the outsourced document collection with an $\mathrm{O}(n)$ communication cost and an impractical $\mathrm{O}(n)$ amount of *cryptographic pairing* computations required at the client side for each query. Shimizu *et. al.* in [139] described how to use the BWT [28] and Pailler's AHE scheme [118] to effectively retrieve the occurrences of a substring. The main drawback of the scheme lies in the significant communication cost, as each query needs to send $\mathrm{O}((m+o_q)\sqrt{n})$ ciphertexts from client to server. Such a cost was reduced by Ishimaki *et. al.* [80] to $\mathrm{O}((m+o_q)\log(n))$, at the price of employing a FHE scheme [60], which introduces a significant constant factor due to the 2 to 3 order of magnitudes ciphertext expansion incurred by FHE schemes. Furthermore, the computational cost for the server is $\mathrm{O}((m+o_q)n\log(n))$, with a large constant overhead (about 10^6) due to the homomorphic computation with FHE ciphertexts.

A multi-user protocol, preserving only the search pattern confidentiality and with the same $\mathrm{O}((m+o_q)\log(n))$ communication cost was proposed in [56]. The main drawbacks of this solution are the need for the client to interact with the data owner to perform a query, and the constraint that only substrings of a fixed length, which must be decided in the SETUP phase of the protocol, can be searched, in turn limiting the impact of the solution. A protocol with a remarkable $\mathrm{O}((m + o_q)z)$ communication cost has recently been proposed in [123]. To the best of our knowledge, this is the first substring search protocol that allows multiple users to perform queries over data coming from multiple data owners with an access control mechanism that allows to restrict, for each document, the users authorized to perform queries. Nonetheless, the protocol is still affected by a lack of access pattern privacy; furthermore, the client must perform a distinct query for each document the client is interested in.

Table 1.1: *Comparison of existing Privacy-Preserving Substring Search protocols with our ones, which are highlighted in gray. In the costs reported, n denotes the size of the document collection $\mathbf{D}$, z the number of documents in $\mathbf{D}$, m the length of the searched substring q, and o_q the number of its occurrences over $\mathbf{D}$.*
[†] *C is a large constant factor due to usage of FHE, e.g., $C \geq 16 \times 10^6$, for 80-bit security parameters*
[‡] *Search and Access pattern privacy*

PPSS Protocol	**Communication Cost**	**Server Cost**	**S. & A.[‡] Privacy**	**Data Owner Off-line**	**Multi User**	**Wildcard Queries**	**Adversary**
[155]	$O(n)$	$O(m \cdot n)$	✓	✓	×	×	Semi-honest
[139]	$O((m+o_q)\sqrt{n})$	$O((m+o_q)n)$	✓	✓	×	×	Semi-honest
[80]	$O(C(m+o_q)\log(n))^{\dagger}$	$O(C(m+o_q)n\log n)^{\dagger}$	✓	✓	×	×	Semi-honest
[56]	$O((m+o_q)\log(n))$	$\Omega((\frac{n}{m}+o_q)\log(n))$	S.✓A.×	×	✓	×	Malicious
[123]	$O((m+o_q)z)$	$O(m \cdot n)$	S.✓A.×	✓	✓	×	Semi-honest
[112]	$\Omega(m\log^5(n)+o_q\log^2(n))$	$\Omega(m\log^5(n)+o_q\log^2(n))$	✓	✓	×	×	Semi-honest
PHE Based	$O(m\log^2(n)+o_q\log(n))$	$O(m \cdot n)$	✓	✓	✓	✓	Semi-honest
ObSQRE	$O(m+o_q)$	$O((m+o_q)\log^3(n))$	✓	✓	×	×	Malicious

Finally, the SA [105] based solutions proposed by Moataz *et. al.* in [112] guarantee the confidentiality of the content of both the substring and the outsourced data, as well as the privacy of the access pattern and the search pattern observed by the server. The access pattern to the outsourced indexing data structures is concealed by employing an ORAM protocol [64]. The asymptotic complexities of the PPSS protocol showed in [112] mainly depends on the size of each document being negligible w.r.t. z, the total number of them. Indeed, it exhibits $O(m\log^3(z))$ communication and computation complexities, assuming that the size of each document is $O(\log^2(z))$. If the size of each document is not negligible w.r.t. z, the computational and communication cost of the solution increase proportionally to the size n of the document collection, by (at least) a factor $\log^2(n)$. Each of the o_q occurrences can be retrieved with o_q accesses to the ORAM, yielding an additional $O(o_q\log^2(n))$ communication cost.

In Tab. 1.1, we compare functionalities and performance of the aforementioned state-of-the-art solutions with our PPSS protocols. We observe that none of existing solutions allows to perform queries containing wildcard characters. We remark that the existing solutions with an asymptotically lower communication cost than our multi-user PPSS protocol based on PHE either do not conceal the access pattern to the untrusted server [56, 123] or hinge upon a FHE scheme [80], which does not allow to achieve the practical performance and bandwidth exhibited by our solution. Instead, ObSQRE is the only PPSS solution achieving an optimal communication cost. Regarding the computational cost, we observe that the only existing PPSS protocol with a sublinear cost [112] has an higher communication cost than our PPSS solution based on PHE and, being based on the ORAM primitive, cannot be trivially extended to enable multiple simultaneous queries from different users with the same security guarantees of our multi-user PPSS solution. We observe that ObSQRE improves both the communication and the computational costs of [112] while retaining the same capabilities and ensuring its security guarantees even against a malicious adversary.

1.4 Backward-Search Algorithm

We now describe the backward-search method [57], an existing algorithm for substring search queries that is the basis for the construction of our privacy-preserving query algorithms employed in our PPSS protocols. The backward-search algorithm relies on

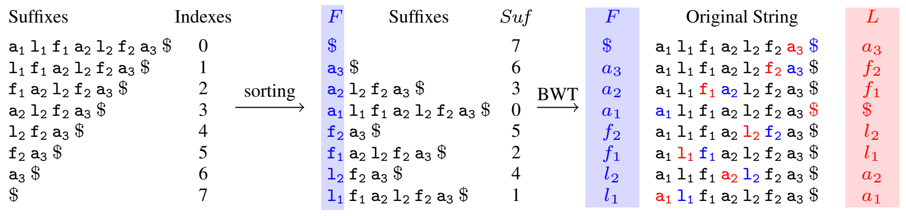

Suffixes
Indexes
$a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ 0
$l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ 1
$f_1\,a_2\,l_2\,f_2\,a_3\,\$$ 2
$a_2\,l_2\,f_2\,a_3\,\$$ 3
$l_2\,f_2\,a_3\,\$$ 4
$f_2\,a_3\,\$$ 5
$a_3\,\$$ 6
$\$$ 7
sorting
F
Suffixes
Suf
$\$$ 7
$a_3\,\$$ 6
$a_2\,l_2\,f_2\,a_3\,\$$ 3
$a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ 0
$f_2\,a_3\,\$$ 5
$f_1\,a_2\,l_2\,f_2\,a_3\,\$$ 2
$l_2\,f_2\,a_3\,\$$ 4
$l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ 1
BWT
F
Original String
L
$\$$ $a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ a_3
a_3 $a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ f_2
a_2 $a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ f_1
a_1 $a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ $\$$
f_2 $a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ l_2
f_1 $a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ l_1
l_2 $a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ a_2
l_1 $a_1\,l_1\,f_1\,a_2\,l_2\,f_2\,a_3\,\$$ a_1

Figure 1.1: *BWT* **L** *and SA* **Suf** *of the string* `alfalfa`

a full-text index based on the BWT [28] and the SA [105] data structures to perform substring search queries with an optimal time complexity of $\mathrm{O}(m + o_q)$. We start with the description of the BWT and the SA data structures, and then we describe in detail the backward-search algorithm.

BWT and SA Data Structures

We consider a string $s \in \Sigma^n$ and a special symbol $\$ \notin \Sigma$, referred to as *end-of-string-delimiter*, which is appended to s. In the following, we denote with an increasing numerical subscript the occurrences of the same character in s (e.g., a_1, a_2 will denote the first and second occurrence of a in s). Given the string s, the *suffix* with *index* i, $i \in \{0, \ldots, n{-}1\}$, is the substring $s[i, \ldots, n]$; the SA of s stores the indexes of all the n suffixes, sorted in lexicographical order w.r.t. an order relationship over $\Sigma \cup \{\$\}$ where $\$$ precedes any character in Σ. The BWT of s is a lossless reversible transformation that makes it more compressible by run-length encoding methods.
The construction of the SA Suf and the BWT L for the string s=`alfalfa` is sketched in Fig. 1.1. The SA is built by applying its definition: first, a list with all the $n{+}1$ suffixes of the string, whose indexes are also computed and stored in a separate array, is built; then, the SA Suf of s is obtained by sorting the array of indexes according to the lexicographical order of the suffixes paired to the indexes. While constructing the SA by naively sorting the suffixes is rather inefficient, there exists a smarter algorithm that allows to build the SA in $\mathrm{O}(n)$ time [82]. The BWT L, highlighted in red in Fig. 1.1, is then defined from the SA as follows: first, the string F, highlighted in blue in Fig. 1.1, is obtained by concatenating the leading characters of the lexicographically sorted suffixes, that is $F[j] = s[Suf[j]]$, $j \in \{0, \ldots, n\}$; then, the j-th character of the BWT L, that is $L[j]$, $j \in \{0, \ldots, n\}$, is the character preceding $F[j]$ in the original string, or $L[j] = \$$ if $F[j]$ is the leading character of s (i.e., if $Suf[j] = 0$). Therefore, the BWT can be computed in $\mathrm{O}(n)$ time directly from the SA as $L[j] = s[(Suf[j]{-}1) \bmod (n{+}1)]$, $j \in \{0 \ldots, n\}$; since the SA can be constructed in $\mathrm{O}(n)$ time, then the BWT can be built in linear time too.

Before delving into the details of the backward-search algorithm, we state and prove three properties about the BWT L and the string F given by the leading characters of the suffixes sorted in lexicographical order; these properties, formalized in Thm. 1.1 are indeed useful to properly explain the backward-search algorithm.

Theorem 1.1. *Consider a string $s \in \Sigma^n$ with $\$ \notin \Sigma$ as trailing character, the BWT L of s, its SA Suf and the the string $F[j] = s[Suf[j]]$, $0{\le}j{\le}n$. Denoting the position of a character $c \in s$ in F and L as $pos_F(c)$ and $pos_L(c)$, respectively, the following properties hold:*

(1) $\forall j \in \{0,\ldots,n\} : pos_L(s[j]) = pos_F(s[(j{+}1) \bmod (n{+}1)])$.

(2) All the occurrences in s of the same character appear in the same order in both F and L, i.e., for each pair of occurrences c_1, c_2 of the same character c in s: $pos_F(c_1){<}pos_F(c_2) \Leftrightarrow pos_L(c_1){<}pos_L(c_2)$.

(3) Consider two occurrences of the same character c in s, denoted by c_1, c_2, such that $pos_L(c_1) < pos_L(c_2)$. If no occurrence c_3 of c such that $pos_L(c_1) < pos_L(c_3) < pos_L(c_2)$ exists, then $pos_F(c_2) = pos_F(c_1) + 1$.

Proof. In the proof of (1), we define an integer $j_i = (Suf[i]{-}1) \bmod (n{+}1)$ for each $i \in \{0,\ldots,n\}$. As the SA Suf has $n{+}1$ distinct values ranging over $\{0,\ldots,n\}$, then all the $n{+}1$ integers j_i are distinct and range over $\{0,\ldots,n\}$. From $L[i] = s[(Suf[i]{-}1) \bmod (n{+}1)]$ and $F[i] = s[Suf[i]]$, $0{\le}i{\le}n$, we derive $L[i] = s[j_i]$ and $F[i] = s[(j_i{+}1) \bmod (n{+}1)]$. Therefore, for any $j_i \in \{0,\ldots,n\}$, $pos_L(s[j_i]) = pos_F(s[(j{+}1) \bmod (n{+}1)]) = i$, which proves (1). In the proof of (2), we denote by j_1 and j_2 the positions in s of c_1 and c_2, i.e., $s[j_1] = c_1$ and $s[j_2] = c_2$, respectively. First, we observe that since F is constructed by concatenating the first characters of the sorted suffixes of s, then $pos_F(c_1) < pos_F(c_2) \Leftrightarrow pos_F(s[j_1{+}1]) < pos_F(s[j_2{+}1])$. Due to (1), $pos_L(c_1) = pos_F(s[j_1{+}1])$ and $pos_L(c_2) = pos_F(s[j_2{+}1])$, thus $pos_F(s[j_1{+}1]) < pos_F(s[j_2{+}1]) \Leftrightarrow pos_L(c_1) < pos_L(c_2)$, which proves (2). Finally, (3) is proven by contradiction. Assume there is no c_3 such that $pos_L(c_1) < pos_L(c_3) < pos_L(c_2)$ with $pos_F(c_2) - pos_F(c_1) \neq 1$. As F contains a sorted sequence of characters in s, having $pos_F(c_2) > pos_F(c_1){+}1$ implies the existence of a further occurrence, c_3, between the two, $pos_F(c_2){>}pos_F(c_3){>}pos_F(c_1)$. Property (2) implies $pos_L(c_2){>}pos_L(c_3){>}pos_L(c_1)$, contradicting the hypothesis. □

Backward-Search Algorithm

The backward search algorithm, outlined in Alg. 1.2, computes the positions of all the occurrences of a string q, given as input, over a string $s \in \Sigma^n$ by hinging upon three data structures: the SA Suf of s, a dictionary `Count` binding a character $c{\in}\Sigma$ to the number of characters smaller than c in the string s (according to the order relation employed to sort suffixes in the SA), and $\tilde{L}$, a full-text index constructed from the BWT L of the string s in order to efficiently compute the RANK procedure.

Definition 1.7 (RANK Procedure). *Given a string $s \in \Sigma^n$ and the full-text index $\tilde{L}$ constructed from the BWT L of s, $\textsc{Rank}(c,i)$ is a deterministic algorithm that, given as input a character $c \in \Sigma$ and an integer $i \in \{0,\ldots,n\}$, employs $\tilde{L}$ to compute the number of occurrences of c in the prefix $L[0,\ldots,i]$ of L, that is $\textsc{Rank}(c,i) = |\{j{\in}\{0,\ldots,i\}\ s.t.\ L[j]{=}c\}|$.*

In the backward-search algorithm reported in Alg. 1.2, we do not detail the RANK procedure, as we will employ different strategies to compute this function in our PPSS protocols, each relying on its own full-text index $\tilde{L}$. Therefore, we now describe the backward-search algorithm by assuming that an implementation of the RANK procedure with $O(T_{rank})$ cost is employed.

As suggested by the name of the algorithm, the backward-search algorithm computes the number o_q of occurrences of the input substring q in the string s by starting

Algorithm 1.2: Backward search for a string $s \in \Sigma^n$

```
Input: $q$: a substring with length $1 \leq m \leq n$
Output: $\mathtt{R_q}$: set of positions in $s$ with leading character of occurrences of $q$
Data: $\tilde{L}$: full-text index constructed from the BWT $L$ of $s$ employed to efficiently compute RANK
      Count: dictionary storing $\forall c \in \Sigma$ the number of chars in $s$ smaller than $c$
      $Suf$: the SA with length $n{+}1$ of the string $s$
1 $\alpha \leftarrow \mathtt{Count}(q[m{-}1])$, $\beta \leftarrow \alpha + \textsc{Rank}(q[m{-}1], n)$, $\mathtt{R_q} \leftarrow \emptyset$
2 for $i \leftarrow m-2$ downto $0$ do
3     $\mathtt{c} \leftarrow q[i]$, $\mathtt{r} \leftarrow \mathtt{Count(c)}$
4     $\alpha \leftarrow \mathtt{r} + \textsc{Rank}(\mathtt{c}, \alpha - 1)$
5     $\beta \leftarrow \mathtt{r} + \textsc{Rank}(\mathtt{c}, \beta - 1)$
6 for $i \leftarrow \alpha$ to $\beta - 1$ do
7     $\mathtt{R_q} \leftarrow \mathtt{R_q} \cup \{Suf[i]\}$
8 return $\mathtt{R_q}$
```

from the last character: specifically, the algorithm identifies the positions, in the string F, of all the occurrences of the character $q[m{-}1]$ in s; as F is a sorted string, then these characters are identified by the substring $F[\alpha, \ldots, \beta - 1]$, where α and β are computed as showed in line 1. Then, in the first iteration of the loop at lines 2-5, the algorithm refines the values α and β to identify the substring of F which contains the leading characters of all the occurrences of the substring $q[m{-}2, \ldots, m{-}1]$ in s. Starting from the occurrences of $q[m{-}1]$ in s (i.e., $F[\alpha, \ldots, \beta - 1]$), the algorithm aims at identifying which of these occurrences are preceded by $q[m{-}2]$: by property (1) in Thm. 1.1, these occurrences can be found by looking at all the occurrences of $q[m{-}2]$ in $L[\alpha, \ldots, \beta{-}1]$. By Property (3) in Theorem 1.1, all these occurrences corresponds to a sequence of consecutive characters (i.e, a substring) of F: if we denote the first of these occurrences as $q[m{-}2]_f$ and the last of them as $q[m{-}2]_l$, then α (resp. β) must be set to $pos_F(q[m{-}2]_f)$ (resp. $pos_F(q[m{-}2]_l){+}1$). As the string F is sorted, the position of $q[m{-}2]_f$ in F can be computed by summing $\mathtt{Count}[q[m{-}2]]$, the number of characters smaller than $q[m{-}2]$ in s, to the number of occurrences of $q[m{-}2]$ preceding $q[m{-}2]_f$ in F; the latter number, by Property (2) in Theorem 1.1, is equal to the number of occurrences of $q[m{-}2]$ preceding $q[m{-}2]_f$ in L, which can be computed as $\textsc{Rank}(q[m{-}2], \alpha{-}1)$, since there are no occurrences of $q[m{-}2]$ preceding $q[m{-}2]_f$ in $L[\alpha, \ldots, n]$. Therefore, by adding $\mathtt{Count}[q[m{-}2]]$ and $\textsc{Rank}(q[m{-}2], \alpha{-}1)$, line 4 correctly updates α to $pos_F(q[m{-}2]_f)$. Similarly, $pos_F(q[m{-}2]_l){+}1$ corresponds to the sum between $\mathtt{Count}[q[m{-}2]]$ and the occurrences of $q[m{-}2]$ in F up to $q[m{-}2]_l$ (including $q[m{-}2]_l$); the latter number can be computed as $\textsc{Rank}(q[m{-}2], \beta - 1)$, as $q[m{-}2]_l$ is the last occurrence of $q[m{-}2]$ in $L[0, \ldots, \beta - 1]$: thus, line 5 correctly updates β to $pos_F(q[m{-}2]_l){+}1$. At the end of this iteration, the number of occurrences of $q[m{-}2, \ldots, m{-}1]$ in s is given by $\beta - \alpha$. We remark that in case there are no occurrences of $q[m{-}2]$ in $L[\alpha, \ldots, \beta - 1]$, then $\textsc{Rank}(q[m{-}2], \alpha - 1) = \textsc{Rank}(q[m{-}2], \beta - 1)$, thus α and β will be updated to the same value.

Each of the subsequent iterations of the loop at lines 2-5 progressively refines α and β to identify all the occurrences of an increasing portion of the substring: that is, in the i-th iteration, $i \in \{1, \ldots, m\}$ the substring $F[\alpha, \ldots, \beta - 1]$ identifies the leading characters of all the occurrences of $q[m{-}i, \ldots, m{-}1]$ in s; eventually, in the last iteration, the values α, β identifies the leading characters of all the occurrences of q in s and the number of such occurrences $o_q = \beta - \alpha$.

The second loop of the algorithm (lines 6-7) retrieves the entries of the SA corresponding to all these occurrences, which contain the positions in the string s of their leading characters: indeed, as $F[i] = s[\mathit{Suf}[i]], i \in \{0, \ldots, n\}$, the position in the original string of the character $F[i]$ is $\mathit{Suf}[i]$. The two loops in Alg. 1.2 perform $m-1$ and o_q iterations, respectively; as each iteration costs $\mathrm{O}(T_{rank})$ and $\mathrm{O}(1)$, respectively, the backward-search has $\mathrm{O}(m \cdot T_{rank} + o_q)$ cost. Since the RANK procedure can be computed with cost $T_{rank} = \mathrm{O}(1)$, employing a specific pre-computed full-text index $\tilde{L}$ with $\mathrm{O}(n \cdot |\Sigma|)$ entries, then the backward-search algorithm locates all the o_q occurrences of a string $q \in \Sigma^m$ over the string s with an optimal $\mathrm{O}(m + o_q)$ cost.

Backward-Search for a Document Collection

A limitation of the backward-search method that hinders its adoption in our scenario is that the algorithm performs substring search queries over a single text instead of a set of strings. To overcome this hindrance, we now describe how to employ the backward-search to efficiently find the repetitions of a substring $q \in \Sigma^m$ over a set of $z \geq 1$ documents $\mathbf{D}=\{D_1, \ldots, D_z\}$. We start by coupling each character of the document collection $\mathbf{D}$ with a a pair of integers $(\mathit{pos}, \mathit{off})$: specifically, for the i-th character of the j-th document, $\mathit{pos}=\sum_{h=1}^{j-1} \mathrm{LEN}(D_j\$)$ and $\mathit{off}=i$. Then, we build a single string s from $\mathbf{D}$ as the ordered concatenation of all the documents, each terminated by an end-of-string delimiter, i.e.: $s=D_1\$D_2\$\ldots D_z\$$, binding to each character of s the pair $(\mathit{pos}, \mathit{off})$ coupled to the corresponding character in $\mathbf{D}$. Therefore, for each character of s, pos is the starting position in s of the document where the character at hand is found, and off is its offset in the said document. Once the string s is built, we compute the three data structures required by the backward search algorithm: the full-text index $\tilde{L}$ constructed from the BWT of s, the dictionary `Count` over the alphabet Σ and the SA Suf of s, where we replace the indexes of the suffixes with the pair of values $(\mathtt{pos}, \mathtt{off})$ coupled to the leading character of each suffix.

When run over these data structures and an input substring q with m characters, Alg. 1.2 computes the set $\mathtt{R_q}$ with o_q pairs $(\mathtt{pos}, \mathtt{off})$; by grouping all the pairs with the same value pos, we obtain, for each document in $\mathbf{D}$, the set of offsets of all the occurrences of q in the document at hand. We remark that from the pair $(\mathtt{pos}, \mathtt{off})$ coupled to the leading character of an occurrence, the position in s of this repetition, if needed, can be trivially computed as $\mathtt{pos}+\mathtt{off}$. Furthermore, in case the application scenario requires also to obtain an identifier of the document where an occurrence is located, then the SA may also store this id for each of its entries.

The backward-search algorithm, when employing these modified data structures, correctly finds all and only the repetitions of q in each of the documents in $\mathbf{D}$ separately. Indeed, any occurrence of q in a document of $\mathbf{D}$ is found in s too; conversely, each occurrence of q in s identifies m characters with no delimiter \$, which correspond to an occurrence of q in a document of $\mathbf{D}$. Thus, this strategy computes the same results as the naive but less efficient solution of running Alg. 1.2 separately over each document.

1.5 Private Information Retrieval

We now introduce the concept of PIR, which, together with the backward-search algorithm, is a fundamental building block of our multi-user PPSS protocol, and we describe the specific PIR employed in our PPSS protocol, that is the Lipmaa's PIR [96].

Definition 1.8 (PIR Protocol)**.** *Consider an untrusted server that stores a public dataset A, which is organized as an array with $e \geq 1$ elements, each of ω bits, and a client with limited computational power. A PIR protocol is a triple of polynomial time algorithms* $(\text{PIR-TRAPDOOR}, \text{PIR-SEARCH}, \text{PIR-RETRIEVE})$*:*

- PIR-TRAPDOOR *is a probabilistic algorithm, executed by the client, that, given an integer $h \in \{0, \ldots, e-1\}$, computes a privacy-preserving representation $\langle h \rangle$, also called trapdoor, and sends it to the server*
- PIR-SEARCH *is a deterministic procedure, run at server side, that, given the privacy-preserving representation $\langle h \rangle$ and the public dataset A, computes a privacy-preserving representation of the entry $A[h]$, i.e., $\langle A[h] \rangle$*
- PIR-RETRIEVE *is a deterministic procedure, run at client side, that extracts the entry $A[h]$ from the privacy-preserving representation $\langle A[h] \rangle$ received from the server*

The privacy guarantees of the PIR protocol ensures that the untrusted server can correctly guess the index h of the element retrieved by the client with probability at most $\frac{1}{e}$, thanks to the usage of privacy-preserving representations for both h and the entry $A[h]$.

We note that to satisfy the privacy guarantees of a PIR protocol, the computational cost at server side must be $\Omega(e)$: indeed, the PIR-SEARCH algorithm must necessarily process all the e entries of A, as otherwise the server trivially learns that the skipped entries are not the ones requested by the client. Conversely, the bandwidth of a PIR protocol should be at most sublinear, as otherwise the protocol is no more efficient than the trivial solution of sending the whole database to the client. In our PPSS protocol, we employ the Lipmaa's PIR because of its low bandwidth. We now describe in detail this PIR protocol.

Lipmaa's PIR Protocol

This PIR protocol hinges upon the DJ LFAHE scheme reported in Section 1.1. To aid the comprehension of the protocol, we start by describing a draft version with no security guarantees. In this protocol, the client and the server read the positions of the cells of the array A with e entries in positional notation with radix $b \geq 2$, i.e., an index h is represented by the sequence of $t{=}\lceil \log_b(e) \rceil$ digits $\{h_0, \ldots, h_{t-1}\}$, with $h_i \in \{0, \ldots, b{-}1\}$ for $i \in \{0, \ldots, t-1\}$, such that $h{=}\sum_{i=0}^{t-1} h_i b^i$. The request of the array element at position h is performed in t communication rounds. First, the client asks the server to select all the cells having the least significant digit of the b-radix expansion of their positions equal to h_0 to compose a new array A_{h_0} concatenating the selected cells in increasing order of their original position, i.e., $A_{h_0}[j]{=}A[j{\cdot}b{+}h_0]$, $0{\leq}j{\leq}\lceil \frac{e}{b} \rceil{-}1$. In the next round, the client asks the server to select the cells in A_{h_0} having the least significant digit of the b-radix expansion of their positions equal to h_1, constructing an array A_{h_1} as $A_{h_1}[j]{=}A_{h_0}[j{\cdot}b{+}h_1] = A[j{\cdot}b^2{+}h_1{\cdot}b{+}h_0]$, $0{\leq}j{\leq}\lceil \frac{e}{b^2} \rceil{-}1$. The next rounds continue employing the subsequent digits of h with the same logic until, in the last round (i.e., the t-th one), a single cell (the h-th one) is identified by the server.

In the proper, fully private, Lipmaa's PIR protocol [96], the client initially generates a keypair (pk, sk) for the DJ LFAHE scheme with a public modulus N, and shares pk with the server. For simplicity of the description, we assume that the entries of the dataset A are small enough to be encrypted in a single ciphertext of length 1 in the DJ scheme (i.e., $\omega \leq \lceil\log_2(N)\rceil$); if this is not the case, then each entry of the dataset can be split in chunks which are small enough to be encrypted in a single ciphertext and then the same trapdoor $\langle h\rangle$ can be employed to independently retrieve each of these chunks.

PIR-TRAPDOOR procedure. The first step of the trapdoor computation considers the value h as the sequence of $t{=}\lceil\log_b(e)\rceil$ digits in b-radix positional representation, for a $b \geq 2$ given as input to the procedure. Each digit h_i with $0{\leq}i{\leq}t{-}1$, is encoded as a bit-string $\texttt{hdigit}_i$, with length b, constructed as $\texttt{hdigit}_i[x]{=}1$ if $x{=}h_i,\ 0$ otherwise, $x{\in}\{0,\dots,b{-}1\}$. Then, each bit $\texttt{hdigit}_i[x]$, $x{\in}\{0,\dots,b{-}1\}$ of the string $\texttt{hdigit}_i$ is encrypted with the DJ LFAHE scheme into a ciphertext with length $l{=}i{+}1$. Thus, the bit-wise encryption of the b-bit string $\texttt{hdigit}_i$ is given as the concatenation of b ciphertexts in $\mathbb{Z}^*_{N^{l+1}}$. The trapdoor $\langle h\rangle$ is returned as the concatenation of the bit-wise encryptions of each b-bit string in the sequence $\texttt{hdigit}_0, \texttt{hdigit}_1, \dots, \texttt{hdigit}_{t-1}$, with total size $b\log_b^2(e)\log(N)$ bits. The trapdoor is then sent to the server, altogether with the radix b employed for its construction. The computational cost of the PIR-TRAPDOOR procedure amounts to $\mathrm{O}(b\log^3(N)\log_b^4(e))$ bit operations, assuming the use of modular multiplication quadratic in the size of the operands.

PIR-SEARCH procedure. The search steps executed at server side follows the t-iterations over the array A reported in the draft description of the PIR protocol. In particular, in the first iteration, the server computes an encrypted array $\langle A_{h_0}\rangle$ with $\lceil\frac{e}{b}\rceil$ items, where each entry $\langle A_{h_0}\rangle[j]$, $0{\leq}j{\leq}\lceil\frac{e}{b}\rceil{-}1$, is a ciphertext in $\mathbb{Z}^*_{N^2}$ encrypting the item $A[j\cdot b + h_0]$ (i.e., $\textsc{Dec}_1(sk, \langle A_{h_0}\rangle[j])$ = $A[j{\cdot}b{+}h_0]$). To this end, each item $\langle A_{h_0}\rangle[j]$ is computed by performing the hybrid dot product, reported in Eq. (1.1), between the sub-array $A[j{\cdot}b,\dots,j{\cdot}b{+}b{-}1]$, whose entries are plaintexts in $\mathbb{Z}_N$, and the bit-wise encryption of the b-bit string $\texttt{hdigit}_0$, whose b ciphertexts are in $\mathbb{Z}^*_{N^2}$. In the second iteration, the server constructs an array $\langle A_{h_1}\rangle$ with $\lceil\frac{e}{b^2}\rceil$ items, where the item $\langle A_{h_1}\rangle[j]$, $0{\leq}j{\leq}\lceil\frac{e}{b^2}\rceil{-}1$, is computed by performing the homomorphic dot product, reported in Eq. (1.2), between the sub-array $\langle A_{h_0}\rangle[j{\cdot}b,\dots,j{\cdot}b{+}b{-}1]$, whose entries are ciphertexts in $\mathbb{Z}^*_{N^2}$, and the bit-wise encryption of the b-bit string $\texttt{hdigit}_1$, whose b ciphertexts are in $\mathbb{Z}^*_{N^3}$. The result of this dot-product is a ciphertext in $\mathbb{Z}^*_{N^3}$ which encrypts the item $\langle A_{h_0}\rangle[j\cdot b + h_1]$. As the latter element is a ciphertext itself, then $\langle A_{h_1}\rangle[j]$ is a *double-layered* encryption of the item $A[j{\cdot}b^2{+}h_1{\cdot}b{+}h_0]$: i.e., $A[j\cdot b^2{+}h_1{\cdot}b{+}h_0] = \textsc{Dec}_1(sk, \textsc{Dec}_2(sk, \langle A_{h_1}\rangle[j]))$. After $t{=}\lceil\log_b(e)\rceil$ iterations, the server computes a single ciphertext $\langle A_{h_{t-1}}\rangle \in \mathbb{Z}^*_{N^{t+1}}$, which is a *t-layered* encryption of the target value $A[h]$. The computational cost of the PIR-SEARCH procedure amounts to $\mathrm{O}(\frac{e}{b}\log^3(N))$ bit operations to compute a ciphertext with $\lceil\log_b(e)\rceil\log(N)$ bits.

PIR-RETRIEVE Procedure. Since $\langle A_{h_{t-1}}\rangle$ is a *t-layered* ciphertext, then the client must remove all these t encryption layers by decrypting t times with decreasing length, i.e., $A[h]{=}\textsc{Dec}_1(sk, \textsc{Dec}_2(sk, \dots, \textsc{Dec}_t(sk, \langle A_{h_{t-1}}\rangle)\ \dots))$. The computational cost of the PIR-RETRIEVE procedure amounts to $\mathrm{O}(\log_b^5(e)\ \log^2(N))$ bit operations to derive the target value $A[h]$. Lastly, the communication cost of the described single-round PIR-protocol amounts to $\mathrm{O}(\log(N) b \log_b^2(e))$ bits sent from client to server, and to $\mathrm{O}(\log(N)\log_b(e))$ bits sent from server to client.

Aggregate PIR Queries. In our PPSS protocol, we also employ an aggregate version of PIR queries, which is able to retrieve multiple consecutive entries from the dataset A with a single trapdoor and the same computational cost at server side. In this version of a PIR protocol, we introduce an additional parameter a, called *aggregation factor*, that specifies the number of entries to be retrieved with a single PIR query; specifically, the dataset A with e elements is logically split in $\lceil \frac{e}{a} \rceil$ chunks of a consecutive entries each, in order to allow the client to retrieve the h-th chunk, $h \in \{0 \ldots, \lceil \frac{e}{a} \rceil - 1\}$, by issuing a single PIR query. In particular, the client generates a PIR trapdoor to retrieve the h-th entry from a dataset with $\lceil \frac{e}{a} \rceil$ entries; the server splits the dataset A in a arrays, with the j-th one, $j \in \{0, \ldots, a-1\}$, containing all the entries $A[i]$ such that $i \bmod a = j$, and employs the unmodified PIR-SEARCH procedure to retrieve the h-th entry from each of these a arrays; these a entries are then sent back to the client. Both the computational and communication costs for the PIR-TRAPDOOR procedure are reduced to, respectively, $\mathrm{O}(b\log^3(N)\log_b^4(\frac{e}{a}))$ and $\mathrm{O}(b\log(N)\log_b^2(\frac{e}{a}))$, as a trapdoor for a dataset with $\lceil \frac{e}{a} \rceil$ entries is generated in place of a trapdoor for a dataset with e entries; conversely, both these costs increase by a factor a in the PIR-RETRIEVE procedure, because a entries are sent back from the server and then decrypted by the client. Remarkably, the computational cost at the server side is unchanged: indeed, the server performs a PIR-SEARCH operations over a dataset with $\lceil \frac{e}{a} \rceil$ entries, thus yielding a $\mathrm{O}(a\frac{e}{ab}\log^3(N)) = \mathrm{O}(\frac{e}{b}\log^3(N))$ cost.

1.6 Oblivious RAM Protocols

In this section, we introduce the concept of ORAM protocol, focusing on ORAMs that arrange the data in a binary tree, because of their polylogarithmic overhead in terms of both bandwidth and computation cost. We detail the inner working of Path ORAM [142] to introduce the main algorithms and data structures of a tree-based ORAM. Following this, we sketch the differences introduced in the Ring [128] and Circuit [156] ORAMs designs.

Path ORAM

Path ORAM splits a dataset of L bits in blocks of B bits and assigns to each one of them a unique identifier, referred to as block id (bid). Although $l = \lceil \frac{L}{B} \rceil$ blocks are sufficient to store the dataset, Path ORAM increases the number of blocks to $M \cdot Z$, where $M = 2^{\lfloor \log_2(l) \rfloor + 1} - 1$ and $Z \geq 1$; these additional blocks, called *dummy*, allow to hide how the l real blocks are scrambled inside the ORAM. The id of dummy blocks is set to a special value $\perp$ to distinguish them from real ones. All the $M \cdot Z$ blocks are partitioned in M *buckets*, each one containing Z blocks; then, the buckets are arranged as a balanced complete binary tree with M nodes, each storing one bucket. Each bucket is encrypted with a semantically secure scheme; a bucket is full if it contains Z real blocks. A Path ORAM tree that stores l=6 blocks is depicted in Fig. 1.2. The $\frac{M+1}{2}$ leaves of the tree are labeled with a leaf id lid, a $\log_2(M+1) - 1$ bits-wide integer that identifies the path of the tree to reach the leaf at hand; specifically, the i-th bit of lid

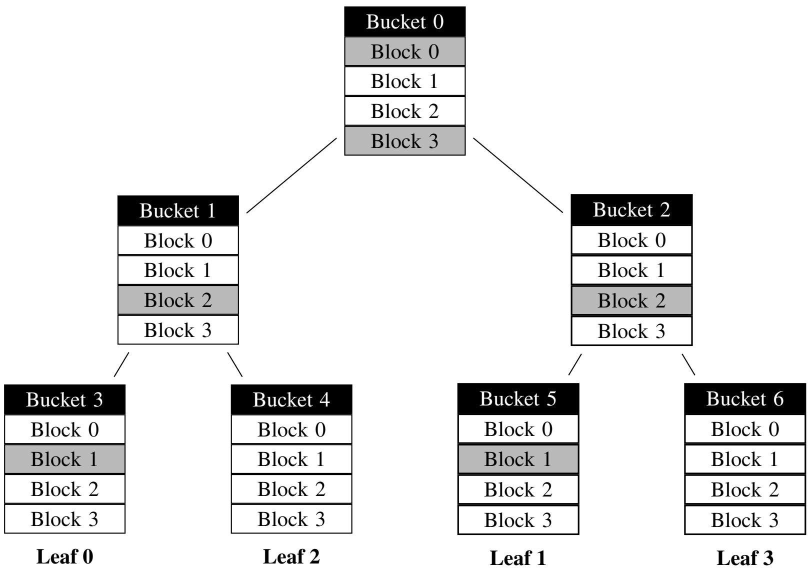


Figure 1.2: *Path ORAM with Z=4 and l=6 (grayed out) real blocks*

(i=0 is the least significant bit) is 0 (resp. 1) if the leaf belongs to the left (resp. right) subtree of the i-th node in the path from the root to the leaf at hand.

To retrieve real blocks from the ORAM, each of them is mapped to a lid, which identifies the path of the tree where the block must reside; this mapping is stored in a data structure called *position map*. Any modification of the tree must preserve this mapping, otherwise blocks cannot be retrieved any longer. All real blocks store their corresponding lid in order to be placed in the proper path. Another data structure, called *stash*, stores the accessed real blocks that have not been pushed back to the ORAM tree yet. The stash analysis of Path ORAM [142] proves that for $Z{\geq}4$ the number of blocks in the stash, denoted with S, is $\mathrm{O}(1)$ with overwhelming probability; thus, the stash can be stored at client side to conceal it from the server.

The content of a specific block from the ORAM can be retrieved through the ACCESS procedure. Given a block id bid, this procedure obtains the leaf id lid corresponding to block bid from the position map, and updates the corresponding entry with a randomly sampled leaf id lid'. Then, it invokes two other procedures: FINDBLOCK(bid, lid, lid') and EVICTION(lid). The former retrieves from the server the whole path containing the leaf with id lid. The client decrypts the fetched path, appends all the real blocks to the stash and looks for the bid block in it. If the block is found, its leaf id is replaced with lid'. FINDBLOCK returns the content of the block, if found, $\perp$ otherwise. The EVICTION procedure writes back the fetched path, with id lid, to the ORAM tree, filling the buckets with as many blocks as possible from the stash. The client computes, for each block in the stash, the deepest bucket of the evicted path that can store the block at hand and, if found, it moves the block from the stash to this bucket. A bucket can store a block with leaf id lid' if it is not full and it belongs to both the evicted path, with id lid, and the path with id lid' (to preserve the property that a block is located along the path corresponding to its leaf id). The eviction stops when there are no more blocks in the stash that can be moved to the evicted path; at this point, the client re-encrypts the path and writes it back to the ORAM. Both procedures

Table 1.2: *Format of the bucket metadata in Ring ORAM. Grayed-out fields must necessarily be encrypted. M is the number of blocks in the ORAM, while Z (resp. D) denotes the maximum (resp. minimum) number of real (resp. dummy) blocks per bucket*

Field	Bit width	Size	Description
IV	λ	1	IV for bucket encryption
Bids	$\log(M{+}1)$	Z	Block ids of real blocks
Lids	$\log(M{+}1)$	Z	Leaf ids of real blocks
Π	$\log(Z{+}D)$	Z	Intra-bucket offset of real blocks
Invalid	1	$Z{+}D$	Blocks already fetched since last bucket write
`cnt`	$\log D$	1	Count accesses to bucket

cost $\mathrm{O}(\log(M){\cdot}Z{\cdot}B)$ on server side and $\mathrm{O}(B(S + \log(M){\cdot}Z))$ on client side, while their bandwidth is $\mathrm{O}(\log(M){\cdot}Z{\cdot}B)$, as the client and the server exchange a whole path. Considering that $B{=}\Omega(\log(M))$, as a block has to store at least the block and the leaf ids, both procedures require $\Omega(\log^2(M){\cdot}Z)$ computational cost and bandwidth.

Path ORAM hides the actual blocks accessed by the client because it introduces a randomly chosen secret mapping between logical block identifiers (i.e., block ids) and the actual access pattern observed by an adversary (i.e., the leaf ids of the paths being fetched). This mapping is randomly updated each time a block is accessed, making consecutive accesses to the same block indistinguishable from accesses to other blocks. The encryption of blocks with a semantically secure cipher hides to the adversary how the blocks are moved in the ORAM tree. To ensure this privacy guarantees, the mapping between block identifiers and the path where the corresponding blocks reside, which is stored in the position map, must be concealed from the server; nonetheless, as the position map has l entries, each of $\log(M)$ bits, it cannot be stored by a client with limited storage capabilities. To overcome this issue, another ORAM, denoted as $ORAM_1$, is employed to store the position map: indeed, if each block of $ORAM_1$ contains up to C entries of the position map, the position map of $ORAM_1$ has $\lceil \frac{l}{C} \rceil$ entries, thus reducing the size of the position map by a factor of C. By recursively applying this strategy to store the position maps of smaller ORAMs, eventually the position map becomes compact enough to be stored at client side. Indeed, by employing $\Theta(\log_C(l))$ recursive ORAMs, the size of the position map of the smallest ORAM becomes $\mathrm{O}(1)$. This recursive strategy introduces a logarithmic factor in both the bandwidth and the computational cost, which become $\mathrm{O}(C{\cdot}B{\cdot}\log^2(M){\cdot}Z)$.

Ring ORAM

Ring ORAM improves over Path ORAM in two ways: the FINDBLOCK procedure achieves a bandwidth of $\mathrm{O}(B{\cdot}\log(M))$ by fetching from the server a single block per bucket instead of the entire bucket; the EVICTION procedure is performed once every $A{\geq}1$ accesses to the ORAM instead of being performed for each access. To fetch a single block per bucket in place of the entire bucket during FINDBLOCK, Ring ORAM enriches the bucket with some metadata (showed in Tab. 1.2), which are employed by the client to choose the blocks to be fetched from each bucket. Specifically, the FINDBLOCK procedure, instead of fetching the buckets along the path containing the searched block, retrieves only their metadata, which is much cheaper as the metadata are much smaller than the entire bucket. Then, for each bucket along this path, FINDBLOCK invokes SELECTOFFSET procedure, which employs the bucket metadata to choose one block to be retrieved from the server, returning its offset in the bucket. The offsets computed by SELECTOFFSET are sent to the server, which retrieves the corresponding blocks from the ORAM tree. As single blocks are retrieved in place of the whole bucket, Ring ORAM mandates to make the decryption of a single block independent from the other ones in the bucket; employing a symmetric key cipher in CounTeR (CTR) mode allows to achieve this goal, also enabling the usage of a single IV for the whole bucket (stored in bucket metadata).

The goal of performing evictions every $A > 1$ accesses instead of being performed for each access is achieved by ensuring that the FINDBLOCK procedure fetches only one real block from the ORAM tree, thus appending only a single block to the stash. To this extent, the SELECTOFFSET procedure selects the offset of the block bid, if found in the bucket, or the offset of a dummy block otherwise. To ensure that there are enough dummy blocks in each bucket to be chosen by the SELECTOFFSET procedure, buckets in Ring ORAM have $Z{+}D$ blocks, where the additional D slots always store dummy blocks. To prevent the adversary from learning if SELECTOFFSET chooses a real or dummy block, all of them are randomly permuted through a permutation Π, stored in the metadata. Furthermore, these two additional properties must be guaranteed: i) each block is fetched from its bucket at most once since the last time the bucket was written back to the ORAM tree; ii) each bucket must be accessed at most D times since the last time it was written to the ORAM tree. The first property is needed to prevent the adversary from distinguishing dummy blocks from real ones from their access frequencies. Indeed, a real block is chosen by SELECTOFFSET procedure, and fetched from a bucket, only when it corresponds to the block bid that must be retrieved by FINDBLOCK, while a dummy block may be chosen in all other cases. To ensure this property, the SELECTOFFSET procedure must always choose a valid block, with a block being marked as invalid in the bucket metadata as soon as it is chosen by SELECTOFFSET. The second property ensures that there are always enough dummy blocks to be chosen in a bucket by SELECTOFFSET: indeed, after D accesses to the bucket, no valid dummy blocks may have left in the bucket. To this extent, the bucket metadata keeps track of the number of accesses to the bucket with a counter `cnt`; when D accesses are reached, a maintenance task called EARLYRESHUFFLE must be invoked. This procedure, upon receiving the Z valid blocks of the bucket, randomly shuffles them with D dummy blocks; then, the bucket is encrypted and written back to the ORAM tree. As the blocks are re-shuffled, they can all be marked as valid, and `cnt` is reset as the bucket has again at least D valid dummy blocks available.

Since only the real block fetched from the ORAM in the FINDBLOCK procedure is appended to the stash, an EVICTION after each FINDBLOCK becomes unnecessary. Therefore, evictions are performed every $A{>}1$ accesses, where A is the *eviction period* of Ring ORAM, whose actual value depends on Z. In order to maximize the average number of blocks evicted from the stash, the paths to be evicted are chosen according to a deterministic schedule, which follows the ids of the paths in increasing order. This schedule guarantees that the overlap between two consecutive evicted paths is limited to the bucket stored in the root node of the ORAM tree, as a bucket at level i of the tree belongs to the evicted path every 2^i consecutive evictions. EVICTION procedure is performed in the same way as in Path ORAM, with the only difference that all the buckets of the evicted path undergo an EARLYRESHUFFLE before being written back to the ORAM, to the extent of randomly permuting real and dummy blocks.

Circuit ORAM

Circuit ORAM is a refinement of Path ORAM tailored for hardware implemented clients, where the server is a large memory on the same machine (or even on the same die). Therefore, this ORAM trades off a low bandwidth for the compactness of the circuit implementing the ORAM client. This is achieved with a simplified EVICTION procedure that evicts at most one block from the stash. The path to be evicted is chosen with the same deterministic schedule of Ring ORAM to minimize the probability that no block can be evicted from the stash. The simplified EVICTION procedure is performed with a single sweep of the evicted path, simultaneously moving at most one block down along this path. The block in the stash that can go deepest in the path is the first block moved down along the path; such block is moved down until either the destination bucket is reached or a block that can go deeper is found in a bucket closer to the root than the destination one. In the first case, the block is placed in the destination bucket unless the bucket is full, while in the second case the block being moved down replaces the block found in the current bucket, which becomes the new block to be moved down along the path. We note that in the first case, if the destination bucket is full, then the block cannot be evicted and has to be placed back in the stash; to reduce the chances of this inconvenience, the EVICTION procedure performs beforehand two sweeps over the metadata (i.e., the block ids and the corresponding leaf ids) of the buckets in the evicted path to compute the additional metadata `dest`. This metadata specifies, for the stash and for each bucket in the evicted path, if a block of the stash/bucket must be moved down along the paths and, if this the case, its destination bucket: indeed, `dest`$[i]$, $i \in \{0,\dots,\log(N{+}1){-}1\}$ (`dest`$[0]$ refers to the stash), stores the bucket where the block of the i-th bucket that can go deepest in the path must be moved, while `dest`$[i]{=}\perp$ if no block from the the i-th bucket must be moved down in the path. The EVICTION procedure then employs `dest` to move these blocks to their destination buckets with a single sweep of the evicted path.

Despite a single block is evicted, the stash growth is limited as FINDBLOCK appends at most the block with id bid to the stash, if found in the fetched path. This path, with the block bid replaced by a dummy one, is re-encrypted and written back to the ORAM tree. To avoid a monotonic growth of the stash in case no blocks from the stash can be evicted, 2 evictions are performed for each access. The additional eviction, although forcing the ORAM to fetch and write back 3 paths per access, allows to keep the stash about 1 order of magnitude smaller than Path and Ring ORAMs.

CHAPTER 2

Comparison-Based Attack Against Noise-Free FHE Schemes

In this chapter, we describe our comparison-based attack against FHE schemes, which allows to recover the plaintext values from ciphertexts, thus completely breaking the FHE scheme at hand. Our plaintext-recovery attack is applicable to any FHE scheme where an efficient algorithm able to determine if a generic ciphertext is the encryption of a fixed plaintext m is available. Since the existence of such algorithm, referred to as m-*distinguisher*, was proven in [158] for linearly-decryptable FHE schemes, then our attack is applicable to any linearly-decryptable FHE scheme, including the two noise-free FHE schemes `OctoM` and `JordanM`. In addition, we show a further attack technique, applicable to any symmetric FHE scheme vulnerable to a key recovery attack when a sufficient number of ciphertexts with known plaintext values is available to the attacker, that allows to mount such attack by relying only on a limited amount of information on the plaintext value of a single ciphertext. Before delving into the details of our attacks, we formalize the notion of m-distinguisher, relying on the *indicator function* over a set of elements:

Definition 2.1 (Indicator Function)**.** *Given a set S and a subset $A \subseteq S$, the indicator function of the elements of A over the ones included in S is defined as: $\mathbb{1}_A : S \to \{0,1\}$, where $\mathbb{1}_A(x) = 1$ if $x \in A$, 0 otherwise.*

Definition 2.2 (m-distinguisher)**.** *Let the four-tuple $(\text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval})$ be an HE scheme with security margin λ, plaintext space $\mathcal{M}$, ciphertext space $\mathcal{C}$ and the keys $\mathtt{k}{=}(sk, pk, evk) \leftarrow \text{KeyGen}(1^\lambda)$. Let $A^m_{\mathtt{k}} \subset \mathcal{C}$, be the set of ciphertexts corresponding to the encryption of a plaintext $m \in \mathcal{M}$, i.e.: $A^m_{\mathtt{k}} = \{c \in \mathcal{C}\,\text{s.t.}\,\text{Dec}(sk, c) = m\}$. Given a plaintext $m \in \mathcal{M}$, an m-distinguisher is a deterministic polynomial time algorithm $\mathtt{A}_m$ taking as input a ciphertext $c \in \mathcal{C}$ and the public portion of $\mathtt{k}$ (i.e.,*

$\mathtt{k}_{\mathtt{pub}} = (pk, evk)$ *for public-key schemes and* $\mathtt{k}_{\mathtt{pub}} = evk$ *for symmetric ones), and computing the indicator function of the elements of* $A^m_{\mathtt{k}}$ *over the set of ciphertexts, namely* $\mathbb{1}_{A^m_{\mathtt{k}}} : \mathcal{C} \to \{0, 1\}$*, in such a way that*

$$\frac{|\{c \in \mathcal{C}\, \text{s.t.}\, \mathtt{A}_m(c, \mathtt{k}_{\mathtt{pub}}) = \mathbb{1}_{A^m_{\mathtt{k}}}(c)\}|}{|\mathcal{C}|} \geq 1 - \epsilon(\lambda),$$

where $\epsilon(\lambda)$ *is a negligible function of the security margin of the system.*

We observe that our definition of m-distinguisher allows the output of the m-distinguisher to be erroneous for a negligible portion $\epsilon(\lambda)$ of ciphertexts. Assuming the existence of such an m-distinguisher for the HE scheme targeted by our attack, we now describe our plaintext-recovery attack.

2.1 Plaintext-Recovery Attack

Before describing our attack, we outline the threat model of our attack, which encompasses the assumptions on the capabilities of the adversary conducting our attack.

Threat Model

The adversary can perform the attack by relying only on publicly-available information. In particular, our attack is performed in a ciphertext-only scenario, which means that the attacker employs only ciphertexts and public portion of the key material (that is, the evaluation key and the public key, if the scheme is asymmetric). Nonetheless, the ciphertext-only scenario generally embodies additional public information that can be inferred from the application domain where the HE scheme is employed. Indeed, since the ciphertexts are involved in a known computation, the corresponding plaintext values are expected to be the input values for this computation. Therefore, we assume that the attacker knows the domain of these input values, which may be extremely smaller than the plaintext domain (e.g., in a power metering application, the input values may be as big as 10^9, while the plaintext domain of an HE scheme is generally the integer ring $\mathbb{Z}_N$, where N may be a much bigger number). Throughout the paper, we denote the domain of the input values as the interval $\overline{D_s} = \{-s+1, \ldots, s+1\}$, where s is an integer representing an upper bound for the majority of the input values (e.g., 10^9 for a power metering application). We remark that, in this definition, there may be input values outside $\overline{D_s}$, what it is relevant for our purposes is that a large portion of input values resides in $\overline{D_s}$.

Attack Strategy

We now detail our plaintext recovery attack against an HE scheme capable of computing the homomorphic greater-than function $HGT_{t,b}$ (defined in Section 1.2), which relies on the existence of an m-distinguisher for the targeted HE scheme.

Throughout the attack, the adversary needs to obtain encryptions of known values: that is, given an integer $h \in \mathbb{Z}_N$, the attacker needs to compute a ciphertext c_h such that $\text{Dec}(sk, c_h) = h$. In case the HE scheme is asymmetric, c_h can be directly obtained by employing the public key encryption algorithm of the scheme, while, in case of a symmetric HE scheme, c_h can be computed from a single encryption of $\hat{m} = 1$ hinging upon the HYBRIDMUL homomorphic operation.

From now on we will assume that, in case our attack is applied to a symmetric HE scheme, an encryption $\hat{c}$ of a unitary plaintext value is available to the attacker (i.e., $\textsc{Dec}(sk, \hat{c}) = 1$), enabling him to obtain encryptions c_h of known values required throughout the attack; at the end of this section, we will show how $\hat{c}$ can be obtained by the adversary.

Comparison-based Attack. The core idea of our attack is to perform a homomorphic binary search over the possible candidates for the value of the plaintext corresponding to the ciphertext at hand. To this end, a comparison function CMP, taking two ciphertexts as inputs and yielding an outcome in cleartext, is computed leveraging the homomorphic greater-than function $HGT_{t,b}$ and the m-distinguisher. In particular, since the result of the $HGT_{t,b}$ function is a ciphertext encrypting either b or $b-1$, by choosing $b = m$ the attacker can employ the m-distinguisher to determine the actual (plaintext) value of $HGT_{t,m}$ (without employing the secret key sk).

Definition 2.3 (Comparison Function)**.** *Consider a FHE scheme with plaintext space $\mathcal{M} = \mathbb{Z}_N$, with $N > 2$, an integer t such that $2 \leq t \leq \frac{q}{2}$, where q is the smallest prime factor of N, and the set of ciphertexts $\mathcal{C}_t = \{c \in \mathcal{C} \, \text{s.t.} \, \texttt{Dec}(sk, c) < t\}$. Given the ciphertexts $c_1, c_2 \in \mathcal{C}_t$ and the m-distinguisher $\texttt{A}_m(c, k_{pub})$ (see Def. 2.2), where m is a fixed plaintext value, $c \in \mathcal{C}$ and k_{pub} is the public portion of the key material of the FHE scheme, the function $CMP : \mathcal{C}_t \times \mathcal{C}_t \rightarrow \{1, 0, -1\}$ is defined as:*

$$CMP(c_1, c_2) = \begin{cases} 1 & \text{if } \texttt{A}_m(HGT_{t,m}(c_1, c_2), \texttt{k}_{\texttt{pub}}) = 1 \wedge \texttt{A}_m(c_1 - c_2 + c_m, \texttt{k}_{\texttt{pub}}) \neq 1, \\ 0 & \text{if } \texttt{A}_m(HGT_{t,m}(c_1, c_2), \texttt{k}_{\texttt{pub}}) = 1 \wedge \texttt{A}_m(c_1 - c_2 + c_m, \texttt{k}_{\texttt{pub}}) = 1, \\ -1 & \text{otherwise} \end{cases}$$

where c_m is an encryption of the plaintext value m (i.e, $\textsc{Dec}(sk, c_m) = m$) computed by the attacker.

Specifically, given the ciphertexts $c_1, c_2 \in \mathcal{C}_t$ with corresponding plaintext values $m_{c_1}, m_{c_2} \in D_t$, $CMP(c_1, c_2) = 1$ if $m_{c_1} > m_{c_2}$, $CMP(c_1, c_2) = 0$ if $m_{c_1} = m_{c_2}$ and $CMP(c_1, c_2) = -1$ if $m_{c_1} < m_{c_2}$: indeed, $\texttt{A}_m(HGT_{t,m}(c_1, c_2), \texttt{k}_{\texttt{pub}}) = 1$ if and only if $m_{c_1} \geq m_{c_2}$, while $\texttt{A}_m(c_1 - c_2 + c_m, \texttt{k}_{\texttt{pub}}) = 1$ if and only if $m_{c_1} = m_{c_2}$.

Denoting with $T_{\texttt{dist}}$ the computational complexity of the m-distinguisher, we have that the time complexity T_{CMP} of CMP is $T_{CMP} = \mathrm{O}(t + 2T_{\texttt{dist}})$, as its execution involves at most two computations of the m-distinguisher plus one computation of the $HGT_{t,m}$ function, which has complexity $\mathrm{O}(t)$. Given a ciphertext c with unknown plaintext, our attack locates the value of its plaintext in the range D_t, computing a ciphertext c_h encrypting a candidate plaintext value h and employing the outcome of $CMP(c, c_h)$ to choose the next candidate plaintext value h' to be tested according to a binary search strategy over the set D_t. Since this set is t elements wide, then the candidate plaintext value is found after $\mathrm{O}(\log(t))$ tests, yielding a computational cost of $\mathrm{O}(T_{CMP} \cdot \log(t)) = \mathrm{O}((t + T_{\texttt{dist}}) \log(t))$.

Starting from the strategy we have just described, we improve its effectiveness extending the range of the recoverable plaintexts. To this end, given a generic ciphertext c, we split the set of recoverable plaintexts into $|D_t| = t$ sized chunks, we find candidate chunks that may contain the plaintext value of c, and we search for it employing the

Algorithm 2.1: Plaintext Recovery Attack

```
    Input: Ciphertext $c \in \mathcal{C}_s$, where $\mathcal{C}_s = \{c \in \mathcal{C}$ s.t. $\textsc{Dec}(sk, c) < s\}$
    Output: Plaintext $m_c = \texttt{Dec}(sk, c)$, $m_c \in \mathbb{Z}_n$
    Data: Ciphertext $\hat{c}$, with $\textsc{Dec}(sk, \hat{c}) = 1$
1   begin
2       for $i \leftarrow 1$ to $\sigma$ do
3           $c_{\texttt{gt}} \leftarrow HGT_{t,m}(c, \textsc{HybridMul}(evk, \hat{c}, (i-1)t))$
4           if $\texttt{A}_{(m)}(c_{\texttt{gt}}, \texttt{k}_{\texttt{pub}}) = 1$ then
5               $c_{\texttt{gt}} \leftarrow \textsc{Add}(evk, HGT_{t,m}(c, \textsc{HybridMul}(evk, \hat{c}, it)), \hat{c})$
6               if $\texttt{A}_{(m)}(c_{\texttt{gt}}, \texttt{k}_{\texttt{pub}}) = 1$ then
7                   $m_c \leftarrow \textsc{BinSearch}(\textsc{Add}(evk, c, \textsc{HybridMul}(evk, \hat{c}, -(i-1)t)))$
8                   if $m_c \neq \perp$ then
9                       return $m_c + (i-1)t$
```

binary search approach in each of the candidate chunks. We denote with D_s the set of recoverable plaintexts ($D_s = \{0, 1, \ldots, s-1\}$, $s \leq N$), and with $\mathcal{C}_s$ the set of ciphertexts obtained encrypting plaintexts in D_s, i.e.: $\mathcal{C}_s = \{c \in \mathcal{C}$ s.t. $\textsc{Dec}(sk, c) < s\}$. The recoverable message space D_s is split into σ chunks containing numerically consecutive plaintexts, with $\sigma = \lceil \frac{s}{t} \rceil$: for instance, the i-th chunk, $1 \leq i \leq \sigma$, contains plaintexts values $\{(i-1)t, \ldots, it-1\}$, while the last one contains values $\{it, \ldots, s-1\}$.

Algorithm 2.1 shows how our improved attack is performed. It iterates over all the σ chunks, testing, for each one, if the unknown plaintext value m_c, corresponding to the input ciphertext c, may be contained in the chunk at hand (lines 2–9). To this end, the algorithm starts by testing if m_c may be in a chunk $\{(i-1)t, \ldots, it-1\}$ by verifying if $GT_{t,m}(m_c, (i-1)t) = m$ (lines 3–4). In case this test succeeds (line 4, case of the `if` being taken), Alg. 2.1 proceeds to test also if m_c is smaller than the upper bound it of the chunk at hand, by verifying that $GT_{t,m}(m_c, it) = m-1$ with an analogous approach (lines 5–6). If the tests at lines 3–6 succeed, then the current chunk may contain the plaintext m_c, and so Alg. 2.1 attempts a plaintext recovery employing the binary search approach described in precedence over the current chunk (line 7). However, the binary search is effective only under the assumption that the sought plaintext is in D_t, thus Alg. 2.1 (line 7) exploits the homomorphic operations to subtract the value of the lower bound $(i-1)t$ of the current chunk from m_c, working on its corresponding ciphertext c, to compute the value of $m_c \bmod t$, which can be retrieved by the binary search strategy.

Nevertheless, we note that the answers of the tests in lines 3–6 are subject to potential false positives. Indeed, if $m_c \notin \{(i-1)t, \ldots, it-1\}$, then $m_c - (i-1) \cdot t \notin \overline{D_t} \vee m_c - i \cdot t \notin \overline{D_t}$: thus, it means that the polynomial $f_{\texttt{sign}}(z) \in \mathbb{Z}_N[z]$, obtained by interpolating points whose z-coordinates range over $\overline{D_t}$ and homomorphically evaluated during the computation of $HGT_{t,m}$, is evaluated on a point $z \notin \overline{D_t}$, hence yielding an outcome which is either outside the set $\{m-1, m\}$ or (by coincidence) inside it. Therefore, it may happen that $f_{\texttt{gt}}(m_c, (i-1)t) = f_{\texttt{sign}}(m_c - (i-1)t) = m$ and $f_{\texttt{gt}}(m_c, it) = f_{\texttt{sign}}(m_c - it) = m-1$ even if $m_c \notin \{(i-1)t, \ldots, it-1\}$. In this case, the chunk $\{(i-1)t, \ldots, it-1\}$ represents a false positive candidate interval for the sought plaintext value m_c. However, these false positive chunks are filtered out later in the algorithm. Indeed, considering the ciphertext fed as input to the BinSearch procedure at line 7 in Alg. 2.1, its corresponding plaintext value $m_c - (i-1)t$ does not belong to the interval D_t if $m_c \notin \{(i-1)t, \ldots, it-1\}$. Since the binary search strategy is effective only under the assumption that the sought plaintext is in D_t, then the binary search will return a valid result (line 8) only if $m_c \in \{(i-1)t, \ldots, it-1\}$, that is when the current chunk is not a false positive. In this case, the actual value of m_c is reconstructed by adding back the lower bound $(i-1)t$ of the current chunk to the value retrieved by the binary search (line 9).

We now consider the time complexity T_a of Alg. 2.1 as a function of the value of the plaintext to be retrieved m_c. Algorithm 2.1 is expected to perform $\lceil \frac{m_c}{t} \rceil$ iterations of the outer loop. Each one of the iterations, save for the last one, will fail the membership tests with very high probability (false positives are unlikely), thus resulting in a computational effort equal to $\mathrm{O}(t + T_{\mathtt{dist}})$, at each iteration. However, we consider the overall worst-case complexity $T_a(m_c)$ of the improved plaintext recovery attack, which assumes that a false positive is found in each iteration:

$$T_a(m_c) = \mathrm{O}(\left\lceil \frac{m_c}{t} \right\rceil (t + T_{\mathtt{dist}} + T_{\mathtt{BinSearch}})) = \mathrm{O}(\log(t)(m_c + \left\lceil \frac{m_c}{t} \right\rceil T_{\mathtt{dist}})) \quad (2.1)$$

Therefore, our attack has a time complexity which is linear in the plaintext value being recovered. This is the main reason why our attack is able to practically recover only ciphertexts whose corresponding plaintext is not too big. However, by setting $t = 2^{20}$ (an upper bound imposed by the $O(t^2)$ computational cost of Lagrange interpolation), we see that, unless $T_{\mathtt{dist}} > 2^{23}$, recovering plaintexts as big as 2^{32} still retains a computational complexity $T_a(m_c) < 2^{40}$. Since many typical FHE scenarios involve computations on relatively small values (e.g., power consumption statistics from smart meters), we deem this plaintext recover capability effective enough to be worth considering. Furthermore, we observe that the linear computational complexity of our attack stems from the fact that the adversary tries in increasing order all the possible plaintext values in the domain D_s, which is a suitable strategy if the general case where tall the elements in D_s are equally likely to be the plaintext value of the given ciphertext; nonetheless, if the attacker, for a specific ciphertext c, can restrict the set of possible plaintext values or it knows that some plaintext values are more likely than others, it can employ a tailored guessing strategy that allows to reduce the number of chunks to be tested before finding the corresponding plaintext value of c.

To conclude the description of our attack, we now show the speed-up obtained by Alg. 2.1 over an exhaustive search strategy leveraging only the m-distinguisher $\mathtt{A}_m$. This latter attack tries all plaintext values $m_c \in \mathbb{Z}_N$ in increasing order, with the recovered plaintext being the first m_c such that $\mathtt{A}_m(\textsc{Add}(evk, c_1, c_2), \mathtt{k_{pub}}) = 1$, with $c_1 = \textsc{HybridMul}(evk, \hat{c}, -m_c)$ and $c_2 = \textsc{Add}(evk, c, c_m)$, where c_m is a ciphertext with corresponding plaintext value equal to m and $\hat{c}$ is the ciphertext with corresponding plaintext $\hat{m} = 1$ which is assumed to be available to the attacker. Denoting the value of the recovered plaintext as m_c, with this strategy we employ the m-distinguisher m_c times, therefore the complexity of this approach is $O(m_c T_{\mathtt{dist}})$. The speed-up of our attack over this simple strategy can be computed as:

$$\frac{m_c T_{\mathtt{dist}}}{T_a(m_c)} = \frac{m_c T_{\mathtt{dist}}}{\log(t)(m_c + \lceil \frac{m_c}{t} \rceil T_{\mathtt{dist}})} = \frac{t T_{\mathtt{dist}}}{\log(t)(t + T_{\mathtt{dist}})} \quad (2.2)$$

This calculation shows that our attack improves the exhaustive search strategy by a constant factor, thus without changing its asymptotic complexity. In particular, the speed-up depends on the values of t, chosen by the attacker, and $T_{\mathtt{dist}}$, given by the

m-distinguisher of the target scheme. Although this improvement may seem negligible, we will show during the evaluation of our attack against the `OctoM` [158] and `JordanM` [158] FHE schemes that the magnitude of the speed-up is significant, as it largely increases the number of recoverable plaintexts.

Relaxing the Assumptions for Comparison-Based Attack. In order to perform our attack, the adversary needs to compute encryptions of known values (e.g., the attacker needs encryptions of the candidate plaintext values while performing the binary search). As already discussed at the beginning of this section, in case the HE scheme is symmetric, such encryptions can be easily computed if a ciphertext $\hat{c}$, whose corresponding plaintext value is $\hat{m} = 1$, is available to the adversary.

We now show how an attacker can easily obtain $\hat{c}$ hinging upon the m-distinguisher and the homomorphic operations. The key observation is that the adversary, given only a generic ciphertext $c \in \mathcal{C}$, can homomorphically evaluate any polynomial $f(z) \in \mathbb{Z}_N[z]$ with no constant term. Indeed, evaluating a polynomial $f(z) = h_{d+1}z^{d+1} + \cdots + h_1 z \in \mathbb{Z}_N[z]$ requires mainly three operations: exponentiations to compute the powers z^i, multiplications between these powers z^i and the coefficients h_i, and the addition of all these terms. The homomorphic evaluation of the polynomial over a generic ciphertext c (i.e., $\textsc{Eval}(evk, f, c)$) can thus be performed by relying on $\textsc{HybridExp}$, $\textsc{HybridMul}$ and $\textsc{Add}$ operations, respectively.

We remark that, for a polynomial with no constant term, $f(0) = 0$ necessarily holds: therefore, in order to compute the ciphertext $\hat{c}$ such that $\texttt{Dec}(sk, \hat{c}) = 1$, the adversary should evaluate the polynomial $f_{\texttt{ne}} \in \mathbb{Z}_N[z]$ such that $f_{\texttt{ne}}(z) = 1$ if and only if $z \neq 0$. This polynomial can be obtained by interpolating the set of N points $\{(0,0), (1,1), \ldots, (N-1, 1)\}$; nevertheless, since interpolating N points would be computationally unfeasible, the adversary can choose an integer $u << N$ such that interpolating the set of $2u - 1$ points $\{(-u+1, 1), \ldots, (0,0), \ldots, (u-1, 1)\}$ becomes computationally feasible. We remark that the usage of $2u - 1$ points instead of N ones has a drawback: the polynomial $f^u_{\texttt{ne}}(z)$ obtained by the interpolation may evaluate to an arbitrary value in $\mathbb{Z}_N$ if $\texttt{abs}(z) \geq u$, where $\texttt{abs}(z)$, for a generic $z \in \mathbb{Z}_N = \{0, \ldots, N-1\}$, is defined as $\texttt{min}(z, N-z)$. Indeed, for a polynomial $f(z)$ obtained by interpolating $2u - 1$ points $\{(x_0, y_0), \ldots, (x_{2u-2}, y_{2u-2})\}$, $f(x_i) = y_i, 0 \leq i < 2u - 1$, holds, but the evaluation $f(z)$ for an integer $z \notin \{x_0, \ldots, x_{2u-2}\}$ is not constrained by the interpolation method. In conclusion, the polynomial obtained by the adversary is:

$$f^u_{\texttt{ne}}(z) = \begin{cases} 0 & \text{if } z = 0 \\ 1 & \text{if } 0 < \texttt{abs}(z) < u \\ \perp & \text{otherwise} \end{cases} \tag{2.3}$$

Here, $\perp$ denotes that the evaluation of $f^u_{\texttt{ne}}(z)$, when $\texttt{abs}(z) \geq u$, may be an arbitrary value in $\mathbb{Z}_N$. This polynomial $f^u_{\texttt{ne}}$ can be homomorphically evaluated by the adversary, since it has no constant term. Given a generic ciphertext c, the adversary computes a ciphertext $c_{\texttt{ne}} = \texttt{Eval}(evk, f^u_{\texttt{ne}}, c)$ and runs the m-distinguisher over the ciphertext $c_m = \textsc{HybridMul}(evk, c_{ne}, m)$: if the m-distinguisher determines that c_m is an encryption of m, the attacker knows that $m \cdot m_{c_{\texttt{ne}}} = m \Rightarrow m_{c_{\texttt{ne}}} = 1$[1], where $m_{c_{\texttt{ne}}} = \textsc{Dec}(sk, c_{\texttt{ne}})$. Therefore, the adversary knows that $c_{\texttt{ne}}$ corresponds to $\hat{c}$, the

[1]The solution of this equation is not necessarily $m_{c_{\texttt{ne}}} = 1$ if the integer m is not coprime with N; however, even in this case, the probability that $m_{c_{\texttt{ne}}} \neq 1$ is a solution is negligible.

required encryption of 1. Once the attacker obtains this encryption $\hat{c}$, then it can compute all the encryptions of arbitrary known values needed to perform the attack. With this method, any ciphertext c whose corresponding plaintext value m_c lies in the interval $\{-u+1, \ldots, -1, 1, \ldots, u-1\}$ would allow the adversary to compute $\hat{c}$.

2.2 Applying Our Attack on Linearly Decryptable FHE Schemes

We now evaluate our attack against two symmetric noise-free FHE schemes [158], `OctoM` and `JordanM`. Since these schemes are linearly-decryptable, there exists an efficient 1-distinguisher for these cryptosystems [158], hence making our attack applicable. Despite the existence of this distinguisher was acknowledged by the authors of `OctoM` and `JordanM`, they claimed the security of their schemes against ciphertext-only adversaries aiming to recover either the plaintext or the secret key [158]; in the following, we show that the confidentiality guarantees of these schemes can be completely subverted by our comparison-based attack. We start by describing both the `OctoM` and the `JordanM` scheme.

Octonion and Jordan Algebrae

We now provide a brief introduction to the two algebras required to understand the FHE schemes evaluated in this work, i.e., the octonion algebra and the Jordan algebra. For a more comprehensive description we refer the reader to [158] or to [6].

Octonion Algebra. The support of the octonion algebra $\mathbb{O}$ is an eight-dimensional vector space over a ring. The FHE schemes we are going to describe instantiate them over the unitary ring $(\mathbb{Z}_N, +, \times), N \in \mathbb{N} \setminus \{0, 1\}$. From now on, we denote the octonion algebra with support $\mathbb{Z}_N^8$ as $\mathbb{O}(\mathbb{Z}_N^8)$. An octonion can be represented as an eight dimensional vector, with the first element being the real component and seven different imaginary components, each corresponding to a different imaginary unit. The sum of two octonions is computed by adding component-wise the elements of the two vectors. The multiplication operation in the octonion algebra, denoted by $*$, is distributive with respect to vector addition, compatible with the scalar multiplication, non commutative and non associative. The multiplicative identity for octonions is the row vector $\mathbb{1} = [1, 0, 0, 0, 0, 0, 0, 0]$, representing the real number 1. An operative description of the octonion multiplication rule is obtained by encoding an octonion a as an 8×8 matrix containing the components of a with proper sign changes. In particular, the multiplication between two octonions a and b can be operatively performed by encoding a in its left 8×8 multiplicative matrix, denoted by A_a^l, and then computing, according to the classic vector-matrix multiplication rule, the product $b \cdot A_a^l$. Similarly to complex numbers, an octonion a has a conjugate, denoted by $\overline{a}$, which can be obtained by flipping the sign of all the imaginary components. Therefore, as for complex numbers, the real part of an octonion a is $\Re(a) = \frac{a+\overline{a}}{2}$, while the imaginary part is $\Im(a) = \frac{a-\overline{a}}{2}$. The product $a * \overline{a}$ is a real number defining the square of the *norm* of an octonion, denoted by $||a||^2$. An octonion $a \in \mathbb{O}(\mathbb{Z}_N^8)$ with $||a|| = 0$ is called *isotropic*. A subspace of $\mathbb{Z}_N^8$ is called *totally isotropic* if all the octonions in it are isotropic. Given an algebra $\mathbb{O}(\mathbb{Z}_N^8)$, an automorphism on this algebra is a linear, bijective mapping $\phi : \mathbb{O}(\mathbb{Z}_N^8) \to \mathbb{O}(\mathbb{Z}_N^8)$ that preserves the product $*$ of the algebra, that is $\phi(a * b) = \phi(a) * \phi(b)$. The set of all automorphisms of an algebra form a group, called the automorphism group. For octonion algebra, the automorphism group is the the exceptional *Lie* group G_2.

Jordan Algebra. The elements of this non commutative and non associative algebra, denoted by $\mathfrak{h}_3(\mathbb{O})$, are 3×3 hermitian matrices of octonions:

$$\alpha = \begin{bmatrix} u & W & V \\ \overline{W} & v & U \\ \overline{V} & \overline{U} & w \end{bmatrix} \tag{2.4}$$

with U, V, W being elements in $\mathbb{O}(\mathbb{Z}_N^8)$ and $u, v, w \in \mathbb{Z}_N$. The compact representation of these matrices is a tuple $\alpha = (u, v, w, U, V, W)$.

The internal composition law of $\mathfrak{h}_3(\mathbb{O})$ is the Jordan product, which is defined as $\alpha \star \beta = \frac{\alpha \cdot \beta + \beta \cdot \alpha}{2}$, where α, β are two matrices and $\cdot$ denotes the usual matrix multiplication. Finally, the determinant of α is $det(\alpha) = uvw - (u\|U\|^2 + v\|V\|^2 + w\|W\|^2) + 2\Re(U * \overline{V} * \overline{W})$, while the automorphism group for Jordan algebra is the exceptional Lie group F_4.

The `OctoM` and `JordanM` cryptosystems

We are now ready to describe the construction of the `OctoM` and `JordanM` cryptosystems.

`OctoM` Cryptosystem. This is a symmetric FHE scheme based on the algebra $\mathbb{O}(\mathbb{Z}_N^8)$, with N being a composite integer.

- **Key Generation.** This algorithm selects the ring $\mathbb{Z}_N$ used as the plaintext space (with N being a composite integer), a totally isotropic subspace V which is closed under octonion multiplication, a generic automorphism ϕ in G_2, and a 8×8 invertible matrix M with entries in $\mathbb{Z}_N$. The secret key k is the tuple $k = (V, \phi, M)$, while the evaluation key is given by the tuple $evk = (N, C_{-1})$, where $C_{-1} = \text{ENC}(k, N - 1)$.

- **Encryption.** Given a plaintext value $m \in \mathbb{Z}_N$, and a key $k = (V, \phi, M)$, this algorithm constructs an octonion $m' = \phi(mi + z)$, where $i = [0, 1, 0, 0, 0, 0, 0, 0]$ is the first imaginary unit and mi is the scalar product between the integer m and i (i.e., $mi = [0, m, 0, 0, 0, 0, 0, 0]$), while $z \in V$ is chosen to make $A^l_{m'}$ (the left multiplication matrix of m') non singular (i.e., $det(A^l_{m'}) \neq 0$). The ciphertext is a matrix $C \in \mathbb{Z}_N^{8\times 8}$ computed as $C = \text{ENC}(k, m) = M^{-1} \cdot A^l_{m'} \cdot M$.

- **Decryption.** Given a ciphertext C, the corresponding plaintext value $m \in \mathbb{Z}_N$ is computed as $m = \text{DEC}(k, C) = \phi^{-1}(\mathbb{1} \cdot (M \cdot C \cdot M^{-1})) \bmod V$. The subspace V modulo operation can be performed by sampling a random vector $v = [v_0, 1, v_2, v_3, v_4, v_5, v_6, v_7]$ in $V^\perp$, the orthogonal space of V, and computing the dot product between v and the octonion $mi + z$, resulting from $\phi^{-1}(\mathbb{1} \cdot (M \cdot C \cdot M^{-1}))$. Indeed, since $v \in V^\perp$ and $z \in V$, then $z \cdot v^T = 0$, therefore the dot product $(mi + z) \cdot v^T = m(i \cdot v^T) = mi$.

- **Homomorphic Addition.** Given two ciphertexts C_1,$C_2 \in \mathbb{Z}_N^{8\times 8}$, the homomorphic addition operation is a simple matrix addition:
$C_{add} = \text{ADD}(evk, C_1, C_2) = C_1 + C_2$.

- **Homomorphic Multiplication.** Given two ciphertexts C_1,$C_2 \in \mathbb{Z}_N^{8\times 8}$, the homomorphic multiplication is performed as follows:
  $C_{mul} = \textsc{Mul}(evk, C_1, C_2) = C_2 \cdot C_1 \cdot C_{-1}$.

We remark that, although `OctoM` was presented in [158] as a FHE scheme, during our evaluation we found out that it is not multiplicatively homomorphic: that is, the homomorphic multiplication operation Mul, given two ciphertexts $C_1, C_2 \in \mathbb{Z}_N^{8\times 8}$, does not fulfill the correctness property (i.e, $\textsc{Dec}(k, \textsc{Mul}(evk, C_1, C_2)) \neq \textsc{Dec}(k, C_1) \times \textsc{Dec}(k, C_2) \bmod N$). Therefore, in order to make our attack applicable to this scheme, we fix the construction of the homomorphic multiplication in order to make it correct; we discuss the design of a correct Mul operation for `OctoM` in Appendix 1.

`JordanM` Cryptosystem. This scheme is a symmetric FHE scheme based on the elements of the Jordan algebra $\mathfrak{h}_3(\mathbb{O})$, with some additional constraints introduced to achieve homomorphic properties.

- **Key Generation.** This algorithm selects the plaintext space $\mathbb{Z}_N$, with N being a composite integer, a random automorphism $\phi \in F_4$, an invertible 3×3 matrix M with entries in $\mathbb{Z}_N$ and three isotropic octonions U,V and W fulfilling $V * \overline{U} = W$ and $\Re(U * \overline{V} * \overline{W}) \neq 0$. The secret key k is the tuple $k = (\phi, M, U, V, W)$, while $evk = N$.
- **Encryption.** Given a plaintext value $m \in \mathbb{Z}_N$, and the key $k = (\phi, M, U, V, W)$, this algorithm first constructs the Jordan algebra element $\alpha_m = (m, v, w, r_U U, r_V V, r_W W)$, with r_U, r_V, r_W, v, w being five random values in $\mathbb{Z}_N$ chosen to make the matrix α_m non-singular; then, the ciphertext $C \in \mathbb{O}(\mathbb{Z}_N)^{3\times 3}$ is computed as $C = \textsc{Enc}(k, m) = M^{-1} \cdot \phi(\alpha_m) \cdot M$.
- **Decryption.** A ciphertext $C \in \mathbb{O}(\mathbb{Z}_N)^{3\times 3}$ is decrypted as $m = \textsc{Dec}(k, C) = \mathbb{1} \cdot \phi^{-1}(M \cdot C \cdot M^{-1}) \cdot \mathbb{1}^T$.
- **Homomorphic Addition.** Given two ciphertexts C_1, $C_2 \in \mathbb{O}(\mathbb{Z}_N)^{3\times 3}$, the homomorphic addition is given by a single matrix addition:
  $C_{add} = \textsc{Add}(evk, C_1, C_2) = C_1 + C_2$.
- **Homomorphic Multiplication.** Given two ciphertexts C_1,$C_2 \in \mathbb{O}(\mathbb{Z}_N)^{3\times 3}$, the homomorphic multiplication is given by a single Jordan product:
  $C_{mul} = \textsc{Mul}(evk, C_1, C_2) = C_1 \star C_2$.

Security Analysis of `OctoM` and `JordanM`

Since `OctoM` and `JordanM` are both linearly decryptable, that is their decryption function is equivalent to the dot product between the ciphertext and the secret key, both conceived as vectors in a d dimensional vector space, known cryptanalytic results applicable to linearly decryptable schemes can be employed against them. Specifically, linearly decryptable schemes can be broken by KPAs: if the attacker knows the corresponding plaintext for d ciphertexts, where d is the dimension of the ciphertext space, then a linear system of equations can be built to recover the key and decrypt any ciphertext. For the two target FHE schemes, the ciphertext space dimension d is 64 for `OctoM`, since a ciphertext is an 8×8 matrix, while $d = 9 \cdot 8 = 72$ for `JordanM`, since a ciphertext is a 3×3 matrix of octonions; thus, a KPA is computationally feasible against such schemes.

Moreover, in [158], authors showed that it is possible to efficiently compute a 1-distinguisher for linearly decryptable FHE schemes. We now describe the construction of such a distinguisher, which is required to apply our attack. Suppose the attacker has a ciphertext C, which can be represented as a d dimensional vector. By hinging upon the homomorphic operations of the FHE scheme, the attacker can compute $d+1$ powers of C, that is $d+1$ ciphertexts $C_i = \textsc{HybridExp}(evk, C, i)$, $i \in \{1, \ldots, d+1\}$. Since the ciphertext space dimension is d, then these $d+1$ ciphertexts are bound to be linearly dependent. Therefore, by definition, there are non trivial solutions to the system of d equations with $d+1$ unknowns a_i defined by $\sum_{i=1}^{d+1} a_i C_i = 0$. Now, since the decryption function is linear and the encryption scheme is multiplicatively homomorphic, it holds the same condition for the corresponding plaintext employing the same coefficients a_i: $\sum_{i=1}^{d+1} a_i m^i = 0$. In case the plaintext m is equal to one, this equation becomes $\sum_{i=1}^{d+1} a_i 1^i = 0 \rightarrow \sum_{i=1}^{d+1} a_i = 0$. Therefore, if we impose the additional constraint that $\sum_{i=1}^{d+1} a_i \neq 0$, a solution to the system of equations $\sum_{i=1}^{d+1} a_i C_i = 0$ cannot be found if and only if $m = 1$. In conclusion, by testing the existence of a solution $a = [a_1, \ldots, a_{d+1}]$ of this system of equations satisfying $\sum_{i=1}^{d+1} a_i \neq 0$, we can determine if $m = 1$ or not. The computational complexity of this 1-distinguisher is $\mathrm{O}(d^3)$, which corresponds to the computational effort needed to solve a system of $\mathrm{O}(d)$ linear equations. We remark that the attacker can solve the system directly on ciphertexts, with no information about the plaintexts or the secret key.

While the existence of the 1-distinguisher and of a KPA is acknowledged by designers of `OctoM` and `JordanM` too [158], their security analysis claims (Thm. 7 of [158]) that, assuming the hardness of solving quadratic equation systems in $\mathbb{Z}_N$ (with composite N), no information about plaintexts can be inferred from corresponding ciphertexts (a notion formalized as *weak-ciphertext only* model). The proof of this claim is based on two reductions: first, the problem of finding the secret key is reduced to the problem of solving a system of multivariate quadratic modular equations; then, the problem of recovering a plaintext is reduced to the problem of solving a univariate quadratic modular equation. Nonetheless, we now show that, by applying our comparison-based attack to the two FHE schemes, we can recover the plaintext value of a generic ciphertext, and, once d plaintexts are recovered, we can compute the secret key by mounting a KPA against these schemes, thus completely subverting the security guarantees of `OctoM` and `JordanM` even in a weak-ciphertext only model.

Attacking `OctoM` and `JordanM`

Once `OctoM` is equipped with a correct homomorphic multiplication operations, such as the one described in Appendix 1, then our comparison based attack is immediately applicable to this scheme, given the existence of the 1-distinguisher and the capability to compute the $HGT_{t,b}$ homomorphic function. We remark that the ciphertext $\hat{c}$ with corresponding plaintext equal to 1 (i.e., $\textsc{Dec}(sk, \hat{c}) = 1$), which is needed in our attack to compute encryptions of arbitrary known values (see Alg. 2.1), can be easily computed with a simpler procedure than the one described in Sec. 2.1 for a generic symmetric FHE scheme: indeed, a ciphertext C_{-1}, whose corresponding plaintext is $N-1$, is embodied in the evaluation key, thus the adversary can simply compute $\hat{C}$ as

$\texttt{Mul}(evk, C_{-1}, C_{-1})$.

Conversely, in order to apply our attack against `JordanM`, we need to cope with the fact that there may be a portion of the ciphertexts on which the outcome of the 1-distinguisher is not correct. In the following paragraphs we analyze which ciphertexts cause the outcome of the 1-distinguisher to be erroneous in `JordanM`, and then we show why these ciphertexts are extremely likely to be computed by the $HGT_{t,1}$ procedure employed in our attack. To solve this issue, we later present a refreshing procedure which allows to overcome the unreliability of the 1-distinguisher in our attack.

$$\begin{cases} m_{mul} = \dfrac{m_1 m_2 + r_{W_1} W * r_{W_2} \overline{W} + r_{V_1} V * r_{V_2} \overline{V}}{2} \\ \qquad + \dfrac{m_2 m_1 + r_{W_2} W * r_{W_1} \overline{W} + r_{V_2} V * r_{V_1} \overline{V}}{2} \\ v_{mul} = \dfrac{r_{W_1} \overline{W} * r_{W_2} W + v_1 v_2 + r_{U_1} U * r_{U_2} \overline{U}}{2} \\ \qquad + \dfrac{r_{W_2} \overline{W} * r_{W_1} W + v_2 v_1 + r_{U_2} U * r_{U_1} \overline{U}}{2} \\ w_{mul} = \dfrac{(r_{V_1} \overline{V} * r_{V_2} V + r_{U_1} \overline{U} * r_{U_2} U + w_1 w_2}{2} \\ \qquad + \dfrac{r_{V_2} \overline{V} * r_{V_1} V + r_{U_2} \overline{U} * r_{U_1} U + w_2 w_1}{2} \end{cases} \tag{2.5}$$

1-**Distinguisher Unreliability**. The structure of the encoding of a message, denoted by α_m, in the `JordanM` scheme is a Jordan algebra element (see Eq. (2.4)) where the 3 octonions are random isotropic octonions and the message is stored as the top left corner of the matrix α_m. In addition, we recall that a generic ciphertext C for the `JordanM` scheme is $C = M^{-1} \cdot \phi(\alpha_m) \cdot M$: therefore, given two ciphertexts $C_1, C_2 \in \mathbb{O}(\mathbb{Z}_N)^{3\times 3}$, their homomorphic product $C_{mul} = \texttt{Mul}(evk, C_1, C_2) = C_1 \star C_2 = M^{-1} \cdot \phi(\alpha_{m_1} \star \alpha_{m_2}) \cdot M$, since the automorphism ϕ preserves the Jordan product.

By considering the Jordan product of these two matrices, namely $\alpha_{mul} = \alpha_{m_1} \star \alpha_{m_2}$, we can see that the elements on the diagonal are the product of the corresponding diagonal elements in the matrices α_{m_1} and α_{m_2}. Indeed, the three elements on the diagonal of the product matrix α_{mul}, denoted by m_{mul}, v_{mul} and w_{mul}, are computed as shown in Eq. (2.5); since U,V, and W are isotropic octonions, then the octonion product with their conjugate is 0, thus all the octonions products in Eq. (2.5) are 0, which means that:

$$\begin{cases} m_{mul} = \frac{1}{2}(m_1 m_2 + m_2 m_1) = m_1 m_2 \\ v_{mul} = \frac{1}{2}(v_1 v_2 + v_2 v_1) = v_1 v_2 \\ w_{mul} = \frac{1}{2}(w_1 w_2 + w_2 w_1) = w_1 w_2 \end{cases}$$

In conclusion, the `JordanM` scheme is multiplicatively homomorphic not only with respect to the message m, but also with respect to the random values v and w chosen during the encoding of the message. This peculiar property has a relevant drawback on the 1-distinguisher: the equation $\sum_{i=1}^{d+1} a_i C_i = 0$ on the $d+1$ powers of a ciphertext C implies not only that $\sum_{i=1}^{d+1} a_i m^i = 0$, but also $\sum_{i=1}^{d+1} a_i v^i = 0$ and $\sum_{i=1}^{d+1} a_i w^i = 0$. Therefore, if at least one of the blinding values v and w are equal to 1, then the 1-distinguisher will determine that the ciphertext C is an encryption of 1 independently from the value of the message m. Nonetheless, this issue is not so relevant for the 1-distinguisher reliability, as the probability of these false positives happening on a randomly selected ciphertext is negligible. Indeed there are N^3 possible unconstrained assignments for m, v and w, and the portion of ciphertexts being affected by this issue is at most $\frac{2N^2}{N^3} < \frac{2}{N}$, which is negligible in the bit size of the modulus N employed in `JordanM`.

1-Distinguisher Unreliability in our Attack. Since `JordanM` is multiplicatively homomorphic with respect to the blinding values v and w, we have that ciphertexts $C \in \mathbb{O}(\mathbb{Z}_n)^{3\times 3}$ computed as $C = HGT_{t,1}(C_x, C_y)$ will have v and w with values in $\{0, 1\}$. As a consequence, invoking the 1-distinguisher on such ciphertexts, as done in our attack, will be providing an incorrect result three times out of four, i.e., in all the cases where either v or w is equal to 1.

To overcome this issue, we devise a ciphertext refreshing procedure, which employs the available ciphertext $\hat{C}$, with corresponding plaintext $\hat{m} = 1$, to compute a new ciphertext $C' = C + C - \hat{C} \star C$, having the same plaintext m, but random values $v' = 2v - \hat{v}v, w' = 2w - \hat{w}w$, where $\hat{v}$ and $\hat{w}$ are the blinding values of ciphertext $\hat{C}$. We note that, if $\hat{v} \neq 1$ (resp. $\hat{w} \neq 1$), then $\texttt{Pr}(v' = 1) = \texttt{Pr}(2 - \hat{v} = v^{-1} \bmod N) < \frac{1}{N}$ (resp. $\texttt{Pr}(w' = 1) < \frac{1}{N}$); therefore, if the 1-distinguisher determines that both C and C' encrypt the plaintext value 1, then the attacker can be sure with overwhelming probability that the outcome is correct, as at least one among C and C' has blinding values different from 1 with overwhelming probability.

We now show that the blinding values $\hat{v}$ and $\hat{w}$ of the ciphertext $\hat{C}$ are equal to 1 with negligible probability, when the adversary employs the method described in Sec. 2.1 to compute the ciphertext $\hat{C}$ for a symmetric FHE scheme. In this method, the attacker homomorphically evaluates the polynomial f^u_{ne}, which is specified in Eq. (2.3). In particular, given a ciphertext C with corresponding plaintext m, the homomorphic evaluation of $f^u_{ne}(m)$ yields a ciphertext $\hat{C}$, whose corresponding plaintext is $\hat{m} = 1$ if $0 < \texttt{abs}(m) < u$ (see Sec. 2.1 for the definition of `abs`). The attacker can determine if $\hat{m} = 1$ by hinging upon the 1-distinguisher. However, since `JordanM` is fully homomorphic with respect to the blinding values v, w too, the polynomial f^u_{ne} is homomorphically evaluated on these random values too: hence, if $0 < \texttt{abs}(v) < u$ (resp. $0 < \texttt{abs}(w) < u$), then $\hat{v} = 1$ (resp. $\hat{w} = 1$). Therefore, in this case the attacker cannot rely on the outcome of the 1-distinguisher, as it will output 1 (i.e, determining that the corresponding plaintext of $\hat{C}$ is 1) independently from the value $\hat{m}$ (thus even if $\hat{m} \neq 1$). Nonetheless, we now show that, if the 1-distinguisher, denoted by $\texttt{A}_1$, outputs 1 on the ciphertext $\hat{C}$ (i.e, if $\texttt{A}_1(\hat{C}) = 1$), then the attacker can employ $\hat{C}$ in the refreshing procedure we described above, since $\texttt{Pr}(\hat{\mathcal{E}} \mid \texttt{A}_1(\hat{C}) = 1)$ is overwhelming, where $\hat{\mathcal{E}} \equiv \hat{m} = 1 \wedge \hat{v} \neq 1 \wedge \hat{w} \neq 1$.

We start by applying Bayes formula and observing that if $\hat{\mathcal{E}}$ holds, then necessarily $\texttt{A}_1(\hat{C}) = 1$:

$$\texttt{Pr}(\hat{\mathcal{E}} \mid \texttt{A}_1(\hat{C}) = 1) = \frac{\texttt{Pr}(\hat{\mathcal{E}} \wedge \texttt{A}_1(\hat{C}) = 1)}{\texttt{Pr}(\texttt{A}_1(\hat{C}) = 1)} = \frac{\texttt{Pr}(\hat{\mathcal{E}})}{\texttt{Pr}(\texttt{A}_1(\hat{C}) = 1)} \tag{2.6}$$

We now introduce three probabilities. We denote by $\hat{p_m}$ the probability that $\hat{m} = 1$ and by $\hat{p_v}$, $\hat{p_w}$ the probabilities that, respectively, $\hat{v} = 1$ and $\hat{w} = 1$. We remark that $\hat{m}$ depends only on the plaintext value m of the original ciphertext C; similarly, the values

$\hat{v}$, $\hat{w}$ depends only on the random values v, w of the ciphertext C. Obviously, the values assigned to m, v, w in the original ciphertext C are all independent among themselves: m is the plaintext value, while v, w are independently sampled from $\mathbb{Z}_N$. Therefore, the values assumed by $\hat{m}$, $\hat{v}$, $\hat{w}$ are independent among themselves too. This fact simplifies the estimation of the probabilities found in Eq. (2.6). Indeed, $\mathtt{Pr}(\hat{\mathcal{E}})$ is equivalent to the probability that $\hat{m} = 1 \wedge \hat{v} \neq 1 \wedge \hat{w} \neq 1$; given the independence among these three conditions, then $\mathtt{Pr}(\hat{\mathcal{E}}) = \hat{p_m}(1 - \hat{p_v})(1 - \hat{p_w})$. Concerning $\mathtt{Pr}(\mathtt{A}_1(\hat{C}) = 1)$, we remark that the 1-distinguisher will output 1 with ciphertext $\hat{C}$ as input if $\hat{m} = 1 \vee \hat{v} = 1 \vee \hat{w} = 1$. Therefore, such probability can be estimated by leveraging the independence of the three conditions as follows:

$$\mathtt{Pr}(\mathtt{A}_1(\hat{C}) = 1) = 1 - \mathtt{Pr}(\hat{m} \neq 1 \wedge \hat{v} \neq 1 \wedge \hat{w} \neq 1) = 1 - (1 - \hat{p_m})(1 - \hat{p_v})(1 - \hat{p_w})$$

By plugging the estimated probabilities in Eq. (2.6), we obtain a formula which depends only on the three probabilities $\hat{p_m}$, $\hat{p_v}$, $\hat{p_w}$:

$$\mathtt{Pr}(\hat{\mathcal{E}} \mid \mathtt{A}_1(\hat{C}) = 1) = \frac{\hat{p_m}(1 - \hat{p_v})(1 - \hat{p_w})}{1 - (1 - \hat{p_m})(1 - \hat{p_v})(1 - \hat{p_w})} \tag{2.7}$$

Thus, we can now estimate these three probabilities to show that $\mathtt{Pr}(\hat{\mathcal{E}} \mid \mathtt{A}_1(\hat{C}) = 1) \approx 1$. For simplicity, we start by estimating $\hat{p_v}$ and $\hat{p_w}$. As already mentioned, since the ciphertext $\hat{C}$ is obtained by homomorphic evaluation of the polynomial f^u_{ne}, then $\hat{v} = f^u_{ne}(v)$ and $\hat{w} = f^u_{ne}(w)$, which means that if $0 < \mathtt{abs}(v) < u$ then $\hat{v} = 1$, and if $0 < \mathtt{abs}(w) < u$ then $\hat{w} = 1$. Since v and w are uniformly sampled from $\mathbb{Z}_N$, then the probabilities that, respectively, $0 < \mathtt{abs}(v) < u$ and $0 < \mathtt{abs}(w) < u$, are both approximately $\frac{u}{N}$. We recall that u is necessarily extremely smaller than the modulus N, since u must be chosen such that interpolating $2u - 1$ points is computationally feasible, while N is an integer sufficiently big to make its factorization computationally unfeasible. Therefore, we obtain that $\hat{p_v} = \hat{p_w} \approx \frac{u}{N} \approx 0$. Concerning $\hat{p_m}$, similarly to $\hat{v}$ and $\hat{w}$, if $0 < \mathtt{abs}(m) < u$ then $\hat{m} = 1$, because $\hat{m} = f^u_{ne}(m)$. However, the probability that $0 < \mathtt{abs}(m) < u$ is much higher than $\frac{u}{N}$. Indeed, we recall that in our threat model we assume that the majority of the input values are assumed to be in the set $\overline{D_s} = \{-s + 1, \ldots, s + 1\}$, with s being generally extremely smaller than the modulus N used for the plaintext space of the targeted FHE scheme. In conclusion, we know that $\hat{p_m} >> \hat{p_v} = \hat{p_w} \approx 0$, which means that $\hat{p_m}(1 - \hat{p_v})(1 - \hat{p_w}) \approx \hat{p_m}$ and $(1 - \hat{p_m})(1 - \hat{p_v})(1 - \hat{p_w}) \approx 1 - \hat{p_m}$. If we plug these approximations in Eq. (2.7), we obtain:

$$\mathtt{Pr}(\hat{\mathcal{E}} \mid \mathtt{A}_m(\hat{C}) = 1) = \frac{\hat{p_m}(1 - \hat{p_v})(1 - \hat{p_w})}{1 - (1 - \hat{p_m})(1 - \hat{p_v})(1 - \hat{p_w})} \approx \frac{\hat{p_m}}{1 - (1 - \hat{p_m})} \approx 1 \tag{2.8}$$

Therefore, when the attacker computes a ciphertext $\hat{C}$ by employing the method proposed in Section 2.1, the probability that this ciphertext does not fulfill the requirement $\hat{\mathcal{E}}$ is negligible; thus, once the attacker obtains $\hat{C}$, it can be used in the refreshing procedure.

In conclusion, the attacker can overcome the unreliability of the 1-distinguisher for the `JordanM` scheme by determining that the corresponding plaintext of a ciphertext C is equal to 1 if and only if $\mathtt{A}_1(C) = 1 \wedge \mathtt{A}_1(C') = 1$, where $C' = C + C - \hat{C} \star C$.

Key Recovery Attack Against Linearly Decryptable FHE Schemes. Since OctoM and JordanM are linearly decryptable FHE schemes, the attacker can easily compute the secret key k with the knowledge of the corresponding plaintext value of d ciphertexts, where d is the number of entries in the secret key vector. Therefore, the adversary can hinge upon our attack to recover the plaintext value of d ciphertexts, and then mount a KPA to recover the secret key, which allows to efficiently decrypt any ciphertext. In case of OctoM and JordanM, the number of plaintexts d that need to be recovered amounts to 64 and 72, respectively, which is surely in the realm of feasibility.

2.3 Computational Complexity and Experimental Evaluation

We now analyze the computational effort required by our attack for the target FHE schemes OctoM and JordanM, by estimating the constant factors in Eq. (2.1) that are specific to the target scheme (i.e., the time complexity of the 1-distinguisher $T_{\texttt{dist}}$). In addition, we show that the speed-up of our attack against the exhaustive search strategy described in Sec. 2.1 is significant for the target FHE schemes. Then, we provide an experimental validation of the performance of our attack, using a prototype implementation targeting OctoM and JordanM schemes. The results confirm that the computational complexity of our attack is linear with the plaintext value being recovered, and show that our attack is more efficient than an exhaustive search strategy. Finally, we analyze the benefits of a parallel implementation of Alg. 2.1, showing that the performance scales linearly with the number of cores employed for the attack.

Performance on Target FHE Schemes

To analyze the computational complexity of our attack against OctoM and JordanM, it is useful to quantify the constant terms in Eq. (2.1), namely t and $T_{\texttt{dist}}$. We recall that the former denotes the set $\{0, \ldots, t-1\}$, chosen by the attacker, that corresponds to the domain of the comparison function $GT_{t,m}$ homomorphically evaluated during the execution of our attack, while the latter parameter $T_{\texttt{dist}}$ denotes the computational complexity of the m-distinguisher, which is specific to the targeted FHE scheme.

By looking at Eq. (2.1), we observe that the computational cost of the attack decreases with higher values of t, which means that the attacker should employ a value t as high as possible. Nonetheless, the computational cost required to interpolate the polynomial computing $GT_{t,m}$ is $\mathrm{O}(t^2)$, hence a choice representing a good trade-off between these computational cost could be $t = 2^{20}$, as observed in Sec. 2.1.

For linearly decryptable schemes, such as OctoM and JordanM, $T_{\texttt{dist}}$, the computational complexity of the 1-distinguisher, is $\mathrm{O}(d^3)$, with $d = 64$ for OctoM and $d = 72$ for JordanM, which means that $T_{\texttt{dist}} = \mathrm{O}(2^{19})$ for both schemes. However, the distinguisher is always invoked twice in our attack, because of the refreshing procedure employed to increase its reliability; therefore the computational complexity we are going to use in place of $T_{\texttt{dist}}$, in the formulae derived in Sec. 2.1 to estimate the computational effort of our attack, is $T'_{\texttt{dist}} = 2T_{\texttt{dist}} = \mathrm{O}(2^{20})$. Given this estimation, we can see that it is practical to recover plaintext values as big as 2^{32}: indeed, the computational effort required to recover a plaintext value $m_c = 2^{32}$ can be computed via Eq. (2.1) by replacing $T_{\texttt{dist}}$ with $T'_{\texttt{dist}} = 2^{20}$ and setting $t = 2^{20}$:

$$T_a(2^{32}) \leq 2^{32}\log(2^{20}) + \left\lceil \frac{2^{32}}{2^{20}} \right\rceil 2^{20}\log(2^{20}) \leq 2^{38}$$

We expect a significant number of FHE ciphertexts to have a corresponding plaintext value smaller than 2^{32}, given the realistic assumption in our threat model on the limited domain $\overline{D_s}$ of input values in FHE-based applications. Therefore, we expect that our attack is able to recover the plaintext value for a significant number of ciphertexts.

We now estimate the cost to recover a plaintext as big as $m_c = 2^{32}$ via an exhaustive search strategy, with the aim of showing the computational savings introduced by our attack strategy. Specifically, the computational cost of the exhaustive search strategy described in Sec. 2.1 for a plaintext $m_c = 2^{32}$ amounts to $\mathrm{O}(m_c T_{\mathtt{dist}}) = 2^{32} \cdot 2^{19} = 2^{51}$ (note that with this strategy we do not need to invoke twice the 1-distinguisher, thus we can use $T_{\mathtt{dist}}$ instead of $T'_{\mathtt{dist}}$). Indeed, the speed-up of our attack, computed by replacing the estimated values for t and $T_{\mathtt{dist}}$ in Eq. (2.2), amounts to:

$$\frac{tT_{\mathtt{dist}}}{\log(t)(t + T'_{\mathtt{dist}})} = \frac{2^{20} \cdot 2^{19}}{20(2^{20} + 2^{20})} > \frac{2^{39}}{2^5 \cdot 2^{21}} = 2^{13}$$

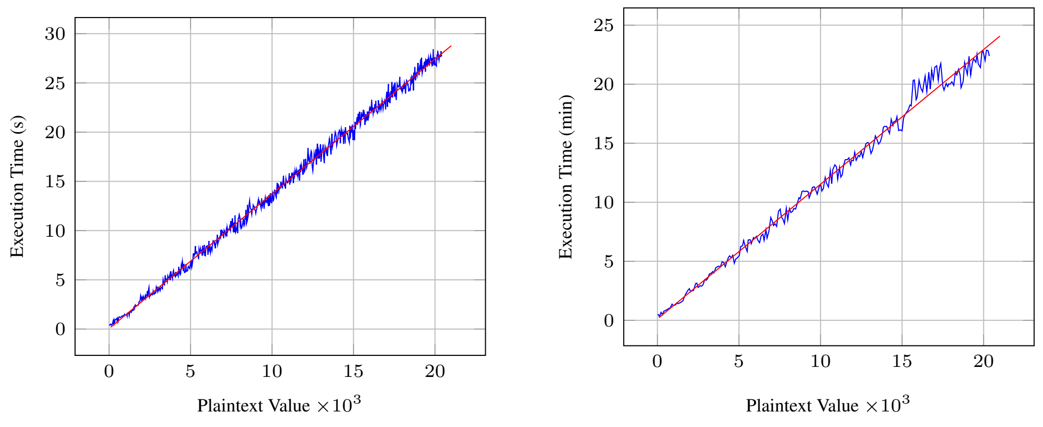
This result shows that the improvement of our attack is not negligible: considering a computational effort fixed a-priori, the number of plaintexts recoverable by our attack is 2^{13} times bigger than the number of plaintexts recoverable by the exhaustive search strategy (when $t = 2^{20}$). For instance, with a computational cost bounded by 2^{38}, our attack can recover plaintexts up to 2^{32}, while the exhaustive search can recover plaintexts up to 2^{19}.

In practice, the security level of the target schemes affects the computational effort to perform the homomorphic operations as well as the modular arithmetic operations needed to evaluate a m-distinguisher. Therefore, such a dependency from the security level and/or the parameter sizes of the cryptoscheme is included in the computational complexity formulae of both our attack and the exhaustive search as the same multiplicative factor (which has been omitted in the previous treatment). Independently from the security margin, when the plaintext values are bounded (e.g., less than 2^{32}) our method largely improves the practicality of their derivation employing only ciphertexts.

Experimental Evaluation

To provide an experimental validation of the effectiveness and performance of our attack, we developed a prototype implementation of the two target FHE schemes(`OctoM` and `JordanM`) and of the described plaintext recovery attack. We realized our implementation leveraging a combination of the `Sage` computer algebra toolkit[2] and of the `numpy` Python module[3]. Such approach allows us to employ highly optimized multiple precision arithmetic primitives at the relatively small cost imposed by the Python bindings. Our prototype implementation was run on a Linux Gentoo server (Gentoo Base System Release 2.6 with kernel version 4.4.95) equipped with two Intel Xeon E5-2620 (8 physical cores each) and 128 GB of DDR-4 DRAM. For all our experiments, we instantiated both `OctoM` and `JordanM` using a small composite integer

[2]`http://www.sagemath.org/index.html`
[3]`http://www.numpy.org/`



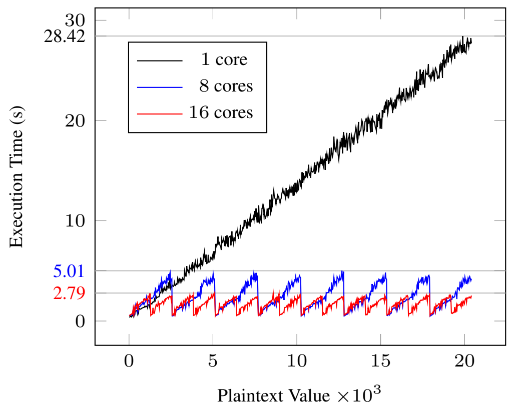
(a) `OctoM` **(b)** `JordanM`

Figure 2.1: *Execution times related to the recovered plaintext value for our attack against* `OctoM` *(a) and* `JordanM` *(b). The red line plots the linear regression models for the datasets:* $y = 1.36 \cdot 10^{-3}\, x + 51.32 \cdot 10^{-3}$ *for* `OctoM`*, and* $y = 1.14 \cdot 10^{-3}\, x + 1.17 \cdot 10^{-1}$ *for* `JordanM`

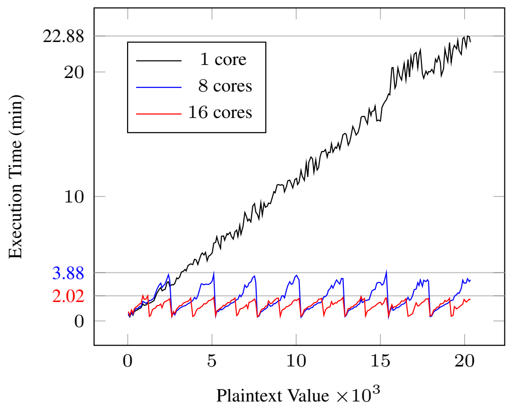
$N = 137 \cdot 149 = 20,413$ for the plaintext space $\mathbb{Z}_N$. We chose to use a small value for N to be able to perform an exhaustive test of the recoverability of all the plaintexts.

Empirical Performance of our Attack. We start our experimental evaluation by analyzing how the time required to perform our attack scales with the plaintext value being recovered. The results of this experiment are shown in Fig. 2.1; specifically Fig. 2.1(a) for `OctoM` and Fig. 2.1(b) for `JordanM`. Both plots confirm the expected linear trend of the computational complexity of our attack with the growth of the plaintext value being recovered. Concerning the execution times, we observe a relevant difference between the two schemes: while our attack allows to recover a plaintext value as big as $2 \cdot 10^4$ in about 30 seconds for an `OctoM` ciphertext, this time is increased to more than 20 minutes to recover the same plaintext value for a `JordanM` ciphertext. The significant difference in the plaintext recovery time is caused by the different arithmetics employed in the two schemes: indeed, in `OctoM` scheme, the homomorphic operations involve matrices in $\mathbb{Z}_N^{8\times 8}$, whose arithmetics is efficiently implemented with native code by `numpy` and `Sage` modules; conversely, in `JordanM`, the homomorphic operations involve matrices with entries over the octonions, whose arithmetics was implemented in pure Python, due to the lack of support in both `Sage` and `numpy` modules.

Performance of a Parallel Implementation. We now investigate the scalability of a parallel implementation of our attack. By looking at Alg. 2.1, we observe that our attack performs a single loop where there are no data dependencies among different iterations. It is thus possible to perform simultaneously these iterations, stopping the computation as soon as one of them finds the correct plaintext. We split evenly the iterations of the algorithm across different processes, and we evaluated the obtained speedups. The execution time of sequential and parallel implementations are reported in Fig. 2.2 (Fig. 2.2(a) for `OctoM` and Fig. 2.2(b) for `JordanM`, respectively). The execution time trends of both `OctoM` and `JordanM` show clearly that the position of the sought plaintext within an iteration determines a variability in the execution time. This is expected as the running time depends on the position of the said plaintext in the



(a) `OctoM`



(b) `JordanM`

Figure 2.2: *Execution times of the parallel implementation of our attack for* `OctoM` *and* `JordanM` *when* 8 *and* 16 *cores are used. The colored numbers on the* y*-axis denotes the maximum execution time of the correspondent implementation*

Table 2.1: *Average speed-up for our implementations and portion of computation running in parallel estimated from average speed-up using the Karp-Flatt metric*

Target Scheme	Cores	Average Speed-Up	Parallel Portion
`OctoM`	8	5.77	0.94
`OctoM`	16	8.45	0.94
`JordanM`	8	6.26	0.96
`JordanM`	16	9.50	0.95

range swept by the process which will find it.

To provide a quantitative evaluation of the scalability of our attack strategy, we follow the approach by Karp and Flatt [83], which proposed a concrete metric to compute the portion of sequential code e in a parallel program starting from the speedup and the number of parallel computing units. We compute the required speed-up for each parallel implementation as the ratio between the average execution time of the attack in the sequential implementation and the average execution time of the attack in the parallel implementation at hand. Table 2.1 reports both the speed-up and the parallel code portion obtained as $1-e$, for an increasing number of cores employed in the attack. We observe that around 95% of the code of our implementations runs in parallel, practically validating the scalability of our approach with the number of processing nodes available.

Speed-up over Exhaustive Search Strategy. We conclude our experimental evaluation by showing that our attack outperforms the exhaustive search strategy described in Sec. 2.1. The speed-up of our attack against this strategy is depicted, for both `OctoM` and `JordanM`, in Fig. 2.3. We recall that the speedup depends on the value of the parameter t chosen during the attack: in particular, the attainable speedup is $\frac{tT_{\mathtt{dist}}}{\log(t)(t+2T_{\mathtt{dist}})}$. The relatively small value for N in our prototype implementation bounds us to pick a small value for t (as Lemma 1.1 mandates that $t \leq \frac{q}{2}$), namely $t = 32$; however, such a

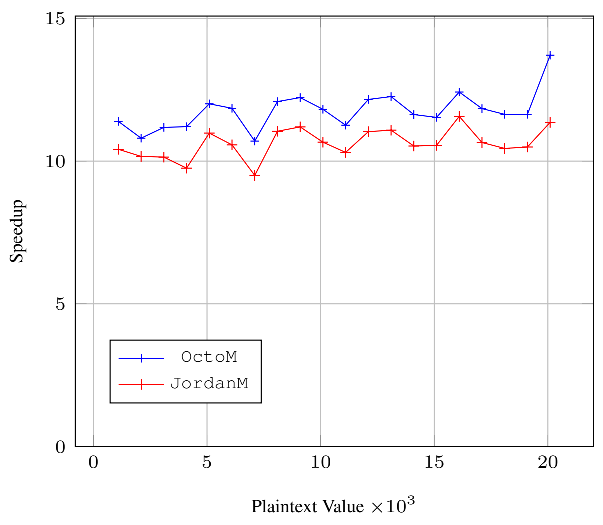


Figure 2.3: *Speed-up obtained by our attack with respect to an exhaustive search for both the* `OctoM` *and the* `JordanM` *cryptosystems. The depicted speedups are obtained picking $t = 32$ as a value for the width of the search intervals*

small value is sufficient to show a practical ten-fold speedup for our plaintext recovery strategy with respect to exhaustive search strategy against both `OctoM` and `JordanM`.

2.4 Mitigating Attack against `OctoM` and `JordanM`

In this section, we propose a simple modification to the two target FHE schemes `OctoM` and `JordanM` that allows to hinder our attack. The main idea is introducing a new non linear operation in the decryption algorithm, making the schemes no longer linearly decryptable over $\mathbb{Z}_N$. This operation is introduced by changing the plaintext space of the two schemes. This plaintext space, for both `OctoM` and `JordanM`, is the integer ring $\mathbb{Z}_N$, with N being a composite number whose factorization is computationally unfeasible. For simplicity, from now on we suppose that N is the product of two prime numbers p, q; however, the same modification can be trivially applied if the integer N has more than two prime factors. As a corollary of the Chinese Remainder Theorem (CRT), the ring $\mathbb{Z}_N$ is isomorphic to $\mathbb{Z}_p \times \mathbb{Z}_q$. Indeed, the CRT defines a bijection φ between these rings, which allows to map each pair of values in $\mathbb{Z}_p \times \mathbb{Z}_q$ to an integer in $\mathbb{Z}_N$, and vice versa. Since these rings are isomorphic, this mapping is a homomorphism: each addition or multiplication done in $\mathbb{Z}_N$ is done at the same time in the rings $\mathbb{Z}_p$ and $\mathbb{Z}_q$. We can hinge upon this isomorphism to build two symmetric FHE schemes, called `OctoPrime` and `JordanPrime`, with plaintext space being the integer ring $\mathbb{Z}_p$, based on, respectively, `OctoM` and `JordanM` schemes. In the following, we show how to construct `OctoPrime` from `OctoM`; `JordanPrime` can be built from `JordanM` in the same way.

- **Key Generation.** Key generation algorithm of `OctoPrime` enriches the secret key k of `OctoM` by including the prime factors of N, namely p and q.
- **Encryption.** Given a plaintext value $m \in \mathbb{Z}_p$ and the secret key k, the encryption algorithm of `OctoPrime` uniformly samples an integer r_q from $\mathbb{Z}_q$ and com-

putes the ciphertext C as $\textsc{Enc}(k, \varphi(m, r_q))$, where $\textsc{Enc}$ denotes the encryption algorithm of `OctoM` and φ denotes the bijection between $\mathbb{Z}_p \times \mathbb{Z}_q$ and $\mathbb{Z}_N$ which can be computed given the prime factors p, q of N.

- **Decryption.** Given a ciphertext c and the secret key k, the decryption algorithm of `OctoPrime` computes the corresponding plaintext $m \in \mathbb{Z}_p$ as $\textsc{Dec}(k, c) \bmod p$, where $\textsc{Dec}$ denotes the decryption algorithm of `OctoM`.

- **Evaluation.** `OctoPrime` employs the same homomorphic operations of `OctoM`.

Despite their simplicity, these modifications are quite important, since `OctoPrime` and `JordanPrime` are no longer linearly decryptable over $\mathbb{Z}_N$: indeed, the last operation of their decryption algorithm is a modular reduction with the modulus p, a prime factor of N, and this operation is not linear in $\mathbb{Z}_N$. Thus, differently from the original FHE schemes, we cannot express the decryption function of `OctoPrime` and `JordanPrime` as a dot product between the ciphertext and the key represented as vectors of a d dimensional vector space with support $\mathbb{Z}_N$, in turn implying that the 1-distinguisher for `OctoM` and `JordanM` cannot be successfully employed for `OctoPrime` and `JordanPrime` schemes. Nonetheless, these schemes are linearly decryptable over $\mathbb{Z}_p$, which implies that there exists a 1-distinguisher for `OctoPrime` and `JordanPrime`. This distinguisher is similar to the one employed for `OctoM` and `JordanM`: it requires to solve the same system of equations, namely $\sum_{i=1}^{d+1} a_i C_i$, with $C_i = \textsc{HybridMul}(evk, C, i)$, but in this case such system must be solved over $\mathbb{Z}_p$ instead of $\mathbb{Z}_N$. However, the attacker does not know the prime factor p (otherwise the schemes would have already been broken) and so the adversary cannot hinge upon such a 1-distinguisher.

Despite the unavailability of a 1-distinguisher for `OctoPrime` and `JordanPrime`, the attacker has another viable strategy, which exploits how these schemes are built. In the following, we analyze this strategy on `OctoPrime` scheme; the same one could be applied to `JordanPrime` too, since they are constructed in the same way. `OctoPrime` can be seen as the composition of two layers: the first one maps the plaintext value $m \in \mathbb{Z}_p$ to an integer $x \in \mathbb{Z}_N$, which is a plaintext value for `OctoM`; then, the integer x is encrypted using the `OctoM` scheme. Therefore, every ciphertext C of `OctoPrime` is a legitimate ciphertext of `OctoM`. Because of this construction, the attacker may try to attack separately these two layers: first, it applies our comparison-based attack on the ciphertext C, retrieving the corresponding plaintext value $x \in \mathbb{Z}_N$ according to `OctoM` encryption; then, it tries to recover the plaintext value $m = x \bmod p$ from the integer x. Luckily, we now show that the first step of this attack is computationally unfeasible. In particular, we recall that our comparison-based attack works by testing if a candidate plaintext value is the corresponding plaintext of a given ciphertext C until a match is found, thus exhibiting a computational complexity which is linear in the number of candidate plaintext values being tested. Since our attack sequentially tests candidate plaintext values in increasing order, it can practically recover only plaintext values not too big (e.g., smaller than 2^{32}). This capability may be sufficient in several FHE-based applications to recover the plaintext values of a large part of ciphertexts, since these values mostly belong to a set $\overline{D_s} = \{-s+1, \dots, s-1\}$, where s is an integer that is expected to be not much bigger than the maximum plaintext value recoverable by our attack. However, in case of `OctoPrime`, our comparison-based attack would not try to recover the actual plaintext value $m \in \mathbb{Z}_p$, but the integer $x = \varphi(m, r_q) \in \mathbb{Z}_N$, where r_q is uniformly sampled from $\mathbb{Z}_q$. Even with plaintext values belonging mostly to D_s, our comparison-based attack would need to test an enormous number of candidate values. In particular, for each plaintext value m, there are q possible values x mapped to it by the bijection φ (one for each value r_q); since there are approximately $\mathrm{O}(s)$ candidate plaintext values to be tested, the attack should find the integer x after testing $\mathrm{O}(qs)$ values, which is computationally unfeasible, as q is a prime factor of N and the prime factors need to be big enough (say, greater than 2^{1024}) to make factoring N computationally hard. In conclusion, our attack can no longer be applied against the modified schemes `OctoPrime` and `JordanPrime`.

In addition, since both the schemes are no longer linearly-decryptable, then they are also no longer vulnerable to the KPA that affects linearly-decryptable schemes. Nonetheless, we now show that it is still possible to mount a refined KPA against `OctoPrime` and `JordanPrime`, which requires the knowledge of the corresponding plaintext value for $d + 1$ ciphertexts, where d is the dimension of the ciphertext space; as in the previous attack strategy, we describe the KPA only for `OctoPrime` scheme, since it can be performed in the same fashion over `JordanPrime`.

KPA Against `OctoPrime`

We have already observed that any ciphertext c for the `OctoPrime` scheme is also a valid ciphertext for the `OctoM` scheme. In particular, since the decryption procedure of `OctoPrime` recovers the plaintext value $m \in \mathbb{Z}_p$ of c by computing $m = \textsc{Dec}(k, c) \bmod p$, where $\textsc{Dec}$ is the decryption function of `OctoM`, then $\textsc{Dec}(k, c)$ is an integer $x \in \mathbb{Z}_N$ such that $x = hp + m$, for an unknown integer $h \in \mathbb{Z}_q$. Therefore, since `OctoM` is linearly decryptable over $\mathbb{Z}_N$, $hp+m = x = \overrightarrow{c} \cdot \overrightarrow{k}^T \bmod N$, where the vectors $\overrightarrow{c} \in \mathbb{Z}_N^d$, $\overrightarrow{k} \in \mathbb{Z}_N^d$ correspond to the representation as $d = 64$ dimensional vectors of the ciphertext c and the secret key k, respectively. To mount the KPA attack, we now assume that the attacker knows the corresponding plaintext values $\{m_1, \dots, m_{d+1}\}$ of $d{+}1$ ciphertexts $\{c_1, \dots, c_{d+1}\}$. For each pair (m_i, c_i), $i \in \{1, \dots, d{+}1\}$, it holds that $\overrightarrow{c_i} \cdot \overrightarrow{k}^T \bmod N = h_i p + m_i$, for an unknown integer $h_i \in \mathbb{Z}_q$, $i \in \{1, \dots, d{+}1\}$. Considering the matrix $C \in \mathbb{Z}_N^{d \times d}$ whose rows are the vectors $\overrightarrow{c_1}, \dots, \overrightarrow{c_d}$, and the vectors $\overrightarrow{m} = [m_1, \dots, m_d] \in \mathbb{Z}_p^d$, $\overrightarrow{h} = [h_1, \dots, h_d] \in \mathbb{Z}_q^d$, the relationships between the d ciphertexts $\{c_1, \dots, c_d\}$ and their corresponding plaintext values $\{m_1, \dots, m_d\}$ can be written in the following matrix form:

$$C \cdot \overrightarrow{k}^T \bmod N = p \cdot \overrightarrow{h}^T + \overrightarrow{m}^T$$

From this equation, we observe that the vector representation $\overrightarrow{k}$ of the secret key k can be written as:

$$\overrightarrow{k}^T = C^{-1} \cdot (p \cdot \overrightarrow{h}^T + \overrightarrow{m}^T) \bmod N = (p \cdot C^{-1} \cdot \overrightarrow{h}^T + C^{-1} \cdot \overrightarrow{m}^T) \bmod N \quad (2.9)$$

where the inverse matrix C^{-1} trivially exists if the d vectors employed to construct C are linearly independent, which is a reasonable assumption in case these vectors represent generic ciphertexts of the `OctoPrime` scheme. We now consider the $d{+}1$-th ciphertext c_{d+1} available to the attacker and its dot product with $\overrightarrow{k}$; by replacing

Eq. (2.9) to $\overrightarrow{k}$ in this dot product, we obtain:

$$\begin{aligned}\overrightarrow{c_{d+1}} \cdot \overrightarrow{k}^T \bmod N &= \overrightarrow{c_{d+1}} \cdot (p \cdot C^{-1} \cdot \overrightarrow{h}^T + C^{-1} \cdot \overrightarrow{m}^T) \bmod N \\ &= (p \cdot \overrightarrow{c_{d+1}} \cdot C^{-1} \cdot \overrightarrow{h}^T + \overrightarrow{c_{d+1}} \cdot C^{-1} \cdot \overrightarrow{m}^T) \bmod N\end{aligned}$$

From this equation, we observe that $\overrightarrow{c_{d+1}} \cdot \overrightarrow{k}^T \bmod N \bmod p = \overrightarrow{c_{d+1}} \cdot C^{-1} \cdot \overrightarrow{m}^T \bmod N \bmod p$, as the other addend of $\overrightarrow{c_{d+1}} \cdot \overrightarrow{k}^T \bmod N$ is a multiple of p. Therefore, since $\overrightarrow{c_{d+1}} \cdot \overrightarrow{k}^T \bmod N \bmod p = \textsc{Dec}(k, c_{d+1}) \bmod p = m_{d+1}$, then $\overrightarrow{c_{d+1}} \cdot C^{-1} \cdot \overrightarrow{m}^T \bmod N \bmod p = m_{d+1}$. We note that, given $d{+}1$ linearly independent ciphertexts $\{c_1, \ldots, c_{d+1}\}$ with known plaintext values $\{m_1, \ldots, m_{d+1}\}$, the attacker can compute the integer $y = \overrightarrow{c_{d+1}} \cdot C^{-1} \cdot \overrightarrow{m}^T \bmod N$; since $y \bmod p = m_{d+1}$, then the attacker can easily recover the modulus p by computing the greatest common divisor between N and $y - m_{d+1}$, as the latter integer is necessarily a multiple of p. Once p is known, the attacker can easily recover the secret key of `OctoPrime`, since the scheme is linearly-decryptable over $\mathbb{Z}_p$.

In conclusion, we show that the secret key of `OctoPrime` can be easily recovered if the attacker obtains $d{+}1 = 65$ (resp. 73 for `JordanPrime`) linearly independent ciphertexts with known plaintext values. In the following, we show our second attack technique, that allows to compute an arbitrary amount of linearly independent ciphertexts with known plaintext values by relying on a very limited information of a single plaintext value.

2.5 Amplifying Known Plainext Attack in FHE Schemes

Our second attack strategy, which hinges upon the homomorphic capabilities of the target FHE scheme, allows to compute an arbitrary amount of linearly independent ciphertexts with known plaintext values, which may be employed to mount a KPA. When applied to FHE schemes vulnerable to a KPA that requires a small enough number of ciphertexts with known plaintext values, our technique allows to recover the secret key of the FHE scheme by relying on a really weak assumption, namely that the attacker knows a set of possible plaintext values for a single ciphertext. We remark that our technique is meaningful only against symmetric FHE schemes, as in an asymmetric one the attacker can easily obtain ciphertexts with known plaintext values by encrypting chosen plaintexts with the public key of the scheme at hand. We now present our attack against a generic FHE scheme with plaintext space $\mathcal{M} = \mathbb{Z}_N$, where N is known to the adversary; then, we show how to tailor our attack to `OctoPrime` and `JordanPrime`, where the attacker does not know the prime p that defines the plaintext space $\mathbb{Z}_p$.

Our attack proceeds in two phases: in the first one, the adversary computes a ciphertext $\tilde{c}$ with a known corresponding plaintext value $\tilde{m} \neq 0$; in the second phase, the attacker employs $\tilde{c}$ to compute as many ciphertexts with known plaintext values as necessary for its purposes (e.g., mounting a KPA). To compute the ciphertext $\tilde{c}$, the adversary relies on the prior knowledge that we assume to perform for our attack: for at least a ciphertext c, the adversary knows the set of possible plaintext values for c. More formally, given a set $S \subset \mathbb{Z}_N$, we define the set $\mathcal{C}_S \subset \mathcal{C}$ as $\{c \in C \mid \textsc{Dec}(k, c) \in S\}$, that is $\mathcal{C}_S$ is the set of ciphertexts whose corresponding plaintext is an integer in S. The attacker can apply our technique to compute the ciphertext $\tilde{c}$ if there is at least one ciphertext c for which the attacker knows the subset S such that $c \in \mathcal{C}_S$. In this case, the attacker can compute the ciphertext $\tilde{c}$ by homomorphically computing a polynomial $f_{S,\tilde{m}}(z)$ defined as:

$$f_{S,\tilde{m}}(z) = \begin{cases} 0 & \text{if } z = 0 \\ \tilde{m} & \text{if } z \in S \\ \perp & \text{otherwise} \end{cases} \tag{2.10}$$

where $\perp$ denotes that the evaluation of the polynomial may be an arbitrary element in $\mathbb{Z}_N$. We observe that the polynomial $f_{S,\tilde{m}}$ must evaluate to 0 when its input is $z = 0$; this constraint is needed as otherwise the polynomial $f_{S,\tilde{m}}$ would have a constant term, and thus it would not be possible to homomorphically compute this polynomial without the availability of a ciphertext with a known corresponding plaintext value. Therefore, if $0 \in S$, then the adversary must homomorphically evaluate the polynomial $f_{S\setminus\{0\},\tilde{m}}$, to avoid an inconsistent definition of such polynomial; nonetheless, this implies that the corresponding plaintext of the ciphertext $\tilde{c} = \text{EVAL}(evk, f_{S\setminus\{0\},\tilde{m}}, c)$, for $c \in \mathcal{C}_S$, is different from $\tilde{m}$ in case the corresponding plaintext value of c is 0, which may happen with a small but non zero probability. Obviously, this issue implies that the attacker should possibly choose a ciphertext $c \in \mathcal{C}_S$ with a known subset S such that $0 \notin S$. Since the adversary has to compute the polynomial $f_{S,\tilde{m}}$ through interpolation, then the subset S must be small enough to allow the adversary to interpolate such a polynomial; we note that this is not a strong limitation on the prior knowledge needed by the adversary to perform the attack, as $f_{S,\tilde{m}}$ can still be computed for reasonably large subsets (e.g., up to 2^{20} elements).

By homomorphically evaluating the polynomial $f_{S,\tilde{m}}$ over a ciphertext $c \in \mathcal{C}_S$, the adversary obtains a ciphertext $\tilde{c}$ with corresponding plaintext value $\tilde{m}$. From this ciphertext, the adversary can now compute an arbitrary number of ciphertexts with known plaintext values. Specifically, to compute $h \geq 1$ ciphertexts $\{c_1, \dots, c_h\}$ with known plaintext values, the adversary chooses at random h univariate non linear polynomials $\{f_1, \dots, f_h\}$ over $\mathbb{Z}_N$ and then it computes the ciphertext $c_i = \text{EVAL}(evk, f_i, \tilde{c})$, $i \in \{1, \dots, h\}$. By the correctness of the homomorphic evaluation (see Def. 1.2), the corresponding plaintext value of c_i is $f_i(\tilde{m})$, which can be easily computed by the attacker. We remark that the set of h ciphertexts will be linearly independent with overwhelming probability, as they are computed as the outputs of randomly chosen non linear functions; the attacker can easily verify if the ciphertexts are linearly dependent and, if this is the case, it can efficiently compute another set of ciphertexts by randomly choosing other non linear polynomials.

We note that the assumption on the information about plaintext values needed by the adversary to perform our attack, namely that the attacker knows a set of possible plaintext values for a single ciphertext c, is generic enough to encompass also application scenarios where different information about plaintext values are available to the adversary. For instance, if an attacker knows that the plaintext values m_1, m_2 of two ciphertexts c_1, c_2 are quite close, then it can apply our technique by assuming that the ciphertext $c_{\texttt{sub}}$, computed as the homomorphic subtraction between c_1 and c_2, belongs to a set $S = \{-u, \dots, u\}$, where u is a reasonable overestimation of the difference between m_1 and m_2. In conclusion, our attack shows that any symmetric FHE scheme that is vulnerable to a KPA, whose secret key was supposed to be recoverable only if the adversary gets to know the corresponding plaintext values for a sufficiently big number of ciphertexts, can be broken with a very limited information about the plaintext value of a single ciphertext. As a consequence, FHE schemes vulnerable to a KPA should never be adopted in real-world applications, even in those scenarios where the information available to the attacker is extremely limited.

Amplifying KPA in `OctoPrime` and `JordanPrime`

In our technique that we have just described, which allows an attacker to compute an arbitrary number of ciphertexts with known plaintexts, we assume that the the target FHE scheme has a plaintext space $\mathcal{M} = \mathbb{Z}_N$, with N known to the adversary. Nonetheless, in `OctoPrime` and `JordanPrime` the plaintext space is $\mathbb{Z}_p$, where p is a secret prime factor of the public modulus N, which implies that the attacker does not know on which ring the plaintext operations are performed. Therefore, to apply our technique to `OctoPrime` and `JordanPrime`, we need to overcome this lack of knowledge.

To this extent, we first analyze the difficulties that arise in our technique because of the lack of knowledge of the plaintext space $\mathbb{Z}_p$. Consider the homomorphic evaluation of the polynomial $f_{S,\tilde{m}}$: the attacker does not know p, therefore it must necessarily construct a polynomial over $\mathbb{Z}_N$. This means that $f_{S,\tilde{m}}(z) \bmod N = \tilde{m}$ if $z \in S$, but when the adversary computes the ciphertext $\tilde{c} = \textsc{Eval}(evk, f_{S,\tilde{m}}, c)$ for a ciphertext $c \in \mathcal{C}_S$, then the ciphertext $\tilde{c}$ is an encryption of $f_{S,\tilde{m}}(z) \bmod p$, as the operations on the plaintext values are performed over $\mathbb{Z}_p$. Since N is a multiple of p, then $f_{S,\tilde{m}}(z) \bmod p = f_{S,\tilde{m}}(z) \bmod N \bmod p = \tilde{m} \bmod p$; therefore, if $\tilde{m} \geq p$, the plaintext value of $\tilde{c}$, that is $\tilde{m} \bmod p$, is different from $\tilde{m}$, in turn leading the adversary to erroneous calculations when employing $\tilde{c}$ to compute the $h \geq 1$ ciphertexts with known plaintext values.

Nonetheless, we recall that the adversary can choose the value $\tilde{m} \in \mathbb{Z}_N$ and then construct the polynomial $f_{S,\tilde{m}}$; therefore, to overcome this issue, the adversary may simply choose an integer $\tilde{m} \in \mathbb{Z}_N$ which is surely less than p, as in this case $\tilde{m} \bmod p = \tilde{m}$. Although the adversary does not know p, it knows that p must be big enough to make the factorization of N computationally infeasible; thus, the adversary can easily estimate a reasonable lower bound for p (e.g., $p \geq 2^{1024}$) and choose an integer $\tilde{m}$ that is much smaller than this lower bound.

A similar issue arises in the second phase of our technique, which relies on the ciphertext $\tilde{c}$ to computes $h \geq 1$ ciphertexts with known plaintext values. Indeed, the adversary can evaluate the randomly chosen non linear polynomials $\{f_1, \ldots, f_h\}$ only over $\mathbb{Z}_N$ instead of $\mathbb{Z}_p$; therefore, when the adversary computes the ciphertext $c_i = \textsc{Eval}(evk, f_i, \tilde{m})$, $i \in \{1, \ldots, h\}$, then c_i is an encryption of $f_i(\tilde{m}) \bmod p$, while the adversary can compute only $f_i(\tilde{m}) \bmod N$. Thus, if $f_i(\tilde{m}) \bmod N \geq p$, then the corresponding plaintext value for c_i, that is $f_i(\tilde{m}) \bmod p$, is different from the one computed by the attacker, that is $f_i(\tilde{m}) \bmod N$. To avoid this issue, the attacker can randomly choose the non linear polynomials $\{f_1, \ldots, f_h\}$ with the additional constraint that their evaluation over $\tilde{m}$ is an integer in $\mathbb{Z}_N$ much smaller than p (i.e., $f_i(\tilde{m}) \bmod N << p$). In this way, the attacker is sure that the plaintext value computed for c_i, namely $f_i(\tilde{m}) \bmod N$, corresponds to the actual one, that is $f_i(\tilde{m}) \bmod p$.

In conclusion, with these modifications, we can apply our attack technique both to `OctoPrime` and to `JordanPrime` FHE schemes, thus recovering their secret key by relying on a very limited amount of information about the corresponding plaintext value of a single ciphertext. Our attack shows that, despite `OctoPrime` and `JordanPrime` are not vulnerable to our comparison-based attack, they cannot be securely employed in any practical application scenario, as the limited amount of information needed by the adversary to apply our technique and mount a KPA can be likely inferred from the application domain.

CHAPTER 3

Multi-User Privacy-Preserving Substring Search Protocol with Polylogarithmic Communication Cost

In this chapter, we describe our multi-user PPSS protocol based on the DJ LFAHE scheme. Our protocol hinges upon the backward-search substring search algorithm, described in Sec. 1.4 and reported in Alg. 1.2. A fundamental building block of this algorithm is the computation of the RANK procedure (see Def. 1.7), which was not detailed in Sec. 1.4 as each of our PPSS protocols employs a different implementation. In this protocol, we consider a RANK procedure that hinges upon the Augmented BWT (ABWT) full-text index. Given a string $s \in \Sigma^n$, the ABWT is constructed from the BWT L of s as follows. Given an integer $P{\geq}1$, referred to as *sample-period*, the ABWT A_P is an array of $\lceil \frac{n+1}{P} \rceil$ entries, where each one is a pair of elements $(rank, l)$; for the i-th entry of A_P, $i \in \{0, \ldots, \lceil \frac{n+1}{P} \rceil{-}1\}$, $A_P[i].rank$ is a dictionary of $|\Sigma|$ entries that binds to a character $c{\in}\Sigma$ the value $\text{RANK}(c, i{\cdot}P{-}1)$, while $A_P[i].l$ is a string of P characters, namely the substring $L[i{\cdot}P, \ldots, (i{+}1){\cdot}P{-}1]$ of the BWT L. Therefore, the RANK procedure, for inputs $c{\in}\Sigma,\ i{\in}\{0, \ldots, n\}$, is easily computed by fetching the $j{=}\lfloor \frac{i}{P} \rfloor$-th entry of the A_P and adding $A_P[j].rank[c]$ to the number of occurrences of c in $A_P[j].l[0, \ldots, i \bmod P]$. The size of each entry of A_P is $O(|\Sigma| \log(n) + P \log(|\Sigma|))$ bits, therefore A_P requires about $O(\frac{|\Sigma|{\cdot}n{\cdot}\log(n)}{P} + n \log(|\Sigma|))$ bits of storage.

We are now ready to describe our multi-user PPSS protocol. We initially present our solution enabling only queries for a substring $q \in \Sigma^m$, for any $m{\geq}1$, and then we discuss further privacy-preserving query algorithms that allow to query also strings containing wildcard characters.

3.1 Our Multi-User PPSS Protocol

The important observation that lead to the design of our PPSS protocol is that both the backward-search algorithm, reported in Alg. 1.2, and the ABWT based RANK procedure perform extremely simple and lightweight operations: indeed, the algorithms either need to retrieve entries from the three data structures being employed, namely the ABWT A_P, the SA Suf and the dictionary `Count`, or they compute simple integer additions. Therefore, the backward-search can be feasibly executed even by a constrained device with limited computational capabilities, such as the client in the scenario considered in our definition of a PPSS protocol. Nonetheless, while the dictionary `Count` can be stored at client side, as it requires only $\mathrm{O}(|\Sigma|\log(n))$ bits of storage, the ABWT and the SA must be necessarily outsourced to a cloud server with significant storage capabilities, as they both require $\Omega(n\log(n))$ bits of storage. Therefore, in order to employ the backward-search method in our PPSS protocol, it is sufficient to allow the algorithm to fetch the needed entries of the outsourced data structures without leaking to the untrusted server any information about which entry is retrieved and its content. In our PPSS protocol we achieve this goal by hinging upon the privacy guarantees of a PIR protocol. Indeed, any two accesses to the same outsourced data structure are indistinguishable for the server, which implies that the server cannot learn anything about such accesses.

We now describe in detail the procedures of our PPSS protocol, which are reported in Alg. 3.1 and Alg. 3.2. In this description, we employ a generic PIR protocol as per Def. 1.8, hereby providing a general PPSS construction combining the backward-search method and any PIR protocol; then, we will analyze the computational and communication costs of our construction when it is instantiated with the Lipmaa's PIR described in Sec. 1.5.

Given the document collection $\mathbf{D} = \{D_1, \dots, D_z\}$ with $z{\geq}1$ documents, the data owner encrypts it with a semantically secure symmetric scheme and outsources it to the remote server. Along with the encrypted version of $\mathbf{D}$, the client computes the privacy-preserving representation $\langle\mathbf{D}\rangle$ by employing the SETUP procedure, reported in Alg. 3.1.

This procedure takes as input the z documents in $\mathbf{D}$ and the security margin λ, an integer representing the computational security level employed to instantiate the underlying cryptographic primitives. The procedure starts with the construction of a single string s obtained concatenating the documents, interleaved with the delimiter \$ (line 2), as required by our strategy to apply the backward-search method to a set of documents (see Sec. 1.4). Subsequently, the algorithm computes the three data structures employed in the backward search algorithm: the SA Suf of the string s; the dictionary `Count` with $|\Sigma|$ entries, that binds to a character $c \in \Sigma$ the number of characters smaller than c in s; the ABWT A_P, constructed from the BWT of s, needed by the RANK procedure.

Then, the data structures that will be outsourced to the server, namely the ABWT A_P and the SA Suf, are cell-wise encrypted (lines 4–7), obtaining arrays $\langle A_P\rangle$ and $\langle Suf\rangle$. To this end, any secure cipher $\mathcal{E}$ can be employed; we choose a symmetric block cipher for efficiency reasons. The algorithms referring to the mentioned cipher are denoted as ($\mathcal{E}$.KEYGEN, $\mathcal{E}$.ENC, $\mathcal{E}$.DEC), where the KEYGEN procedure yields a pair of public and private keys, i.e.: $pk_{\mathcal{E}}$, $sk_{\mathcal{E}}$ (line 3), where $pk_{\mathcal{E}} = sk_{\mathcal{E}}$ if $\mathcal{E}$ is a symmetric-key cipher.

Algorithm 3.1: SETUP Procedure of our PPSS Protocol

```
Function SETUP($\mathbf{D}$,$\lambda$):
    Input: Document Collection $\mathbf{D} = \{D_1, \dots, D_z\}$, security parameter $\lambda$
    Output: $\langle\mathbf{D}\rangle$, privacy-preserving representations of $\mathbf{D}$;
            $aux_s$, secret auxiliary information employed by the client to perform search requests
    Data: $P$, sample period employed to construct the ABWT $A_P$
1   begin
2       $s \leftarrow$ CONCAT$(D_1, \$, D_2, \$, \dots, D_z, \$)$, $n \leftarrow \sum_{i=1}^{z}$ LEN$(D_i) + 1$
        /* Compute the SA $Suf$, the ABWT $A_P$ and the Count dictionary for
           string $s$ (see Section 1.4)                                      */
3       $(pk_{\mathcal{E}}, sk_{\mathcal{E}}) \leftarrow \mathcal{E}$.KEYGEN$(\lambda)$
4       for $i \leftarrow 0$ to $n$ do
5           $\langle Suf\rangle[i] \leftarrow \mathcal{E}$.ENC$(pk_{\mathcal{E}}, Suf[i])$
6       for $i \leftarrow 0$ to $\lceil \frac{n+1}{P} \rceil - 1$ do
7           $\langle A_P\rangle[i] \leftarrow \mathcal{E}$.ENC$(pk_{\mathcal{E}}, A_P[i])$
8       $aux_s \leftarrow$ (Count, $sk_{\mathcal{E}}$)
9       $\langle\mathbf{D}\rangle \leftarrow (\langle A_P\rangle, \langle Suf\rangle)$
10      return $(aux_s, \langle\mathbf{D}\rangle)$
```

At line 8, the secret information kept by the client aux_s is computed as the dictionary Count and the secret key of cipher $\mathcal{E}$. Finally, the SETUP procedure in Alg. 3.1 returns the secret data to be kept by the client, aux_s=(Count, $sk_{\mathcal{E}}$), and the privacy-preserving representation $\langle\mathbf{D}\rangle = (\langle A_P\rangle, \langle Suf\rangle)$ of the document collection, which is outsourced, altogether with the encrypted documents, to the remote server. The SETUP procedure requires $O(n)$ bit operations, mostly due to the computation and the encryption of the ABWT A_P and the SA Suf; outsourcing the privacy-preserving representation $\langle\mathbf{D}\rangle$ requires to upload and store on the remote server $\mathrm{O}(n\log(n))$ bits.

Algorithm 3.2: QUERY Procedure of our PPSS Protocol

```
Function QUERY($q$, $aux_s$, $\langle\mathbf{D}\rangle$):
    Input: $q$, $m$-character string to be searched;
           $aux_s$, secret auxiliary information available to the client containing (Count, $sk_{\mathcal{E}}$);
           $\langle\mathbf{D}\rangle$, remotely accessed privacy-preserving representation of the document collection $\mathbf{D}$,
           containing $(\langle A_P\rangle, \langle Suf\rangle)$.
    Output: Set of positions of occurrences of $q$ in $\mathbf{D}$
1   begin
2       $\alpha \leftarrow$ Count$(q[m-1])$, $\beta \leftarrow \alpha +$ RANK$(q[m-1], n)$ // start of the 1st phase: Qnum
3       for $i \leftarrow m-2$ downto $0$ do
4           c $\leftarrow q[i]$, r $\leftarrow$ Count(c)
5           $\alpha \leftarrow$ r + RANK(c, $\alpha - 1$)
6           $\beta \leftarrow$ r + RANK(c, $\beta - 1$)
7       return BATCHEDRETRIEVAL$(\alpha, \beta-1, aux_s, \langle SA\rangle)$
```

The QUERY procedure, reported in Alg. 3.2, takes as input the m-character string to be searched q, the secret parameters of the client aux_s= (Count, $sk_{\mathcal{E}}$), and the privacy-preserving representation $\langle\mathbf{D}\rangle = (\langle A_P\rangle, \langle Suf\rangle)$ outsourced at server side.

The operations performed during the execution of the QUERY procedure are grouped in two phases. The first phase, labeled as Qnum (lines 2–6), allows to evaluate as $o_q = \beta - \alpha$ the total number of occurrences of q in the remotely stored documents. We observe that, since the dictionary Count is available to the client, then the only difference between this phase of our QUERY procedure and the corresponding phase of

Algorithm 3.3: RANK Procedure employed in the QUERY procedure of our PPSS Protocol

```
Function RANK($c$,$i$):
    Input: $c$, character in the alphabet $\Sigma$
           $i$, integer in $\{0,\dots,n\}$
    Output: ctr: number of occurrences of $c$ in $L[0,\dots,i]$
    Data: $P$, sample period employed to construct the ABWT $A_P$
          $\langle A_P\rangle$, the privacy-preserving representation of the ABWT $A_P$ of the document collection $\mathbf{D}$
          $sk_{\mathcal{E}}$, secret key of the semantically secure cipher $\mathcal{E}$
1   begin
2       $h \leftarrow \lfloor \frac{i}{P} \rfloor$
3       $\langle h\rangle \leftarrow$ PIR-TRAPDOOR($h$)
4       ctx $\leftarrow$ PIR-SEARCH($\langle h\rangle, \langle A_P\rangle$) /* executed at server side */
5       entry $\leftarrow$ PIR-RETRIEVE(ctx) /* entry $= \langle A_P[h]\rangle$ */
6       entry $\leftarrow \mathcal{E}$.DEC($sk_{\mathcal{E}}$, entry) /* entry $= A_P[h]$ */
7       ctr $\leftarrow$ entry.$rank[c]$
8       for $j \leftarrow 0$ to $i \bmod P$ do
9           if entry.$l[j] = c$ then
10              ctr $\leftarrow$ ctr$+1$
11      return ctr
```

the backward-search algorithm (lines 1–5 in Alg. 1.2) is the computation of the RANK procedure, which is reported in Alg. 3.3. This algorithm, given a character $c \in \Sigma$ and an integer $i \in \{0,\dots,n\}$, first privately fetches the $h = \lfloor \frac{i}{P} \rfloor$-th entry of the cell-wise encrypted array $\langle A_P\rangle$ from the remote server by hinging upon a PIR protocol (lines 2–5); then, it decrypts the retrieved entry (line 6), which corresponds to $A_P[h]$, and it computes ($Rank(c,i)$) as the sum between $A_P[h].rank[c]$ and the number of occurrences of c in $A_P[h].l[0,\dots,i \bmod P]$ (lines 7–11).

Algorithm 3.4: BATCHEDRETRIEVAL Procedure employed in the QUERY procedure of our PPSS Protocol

```
Function BATCHEDRETRIEVAL (first,last,$aux_s$,$\langle A\rangle$):
    Input: first, last: starting and ending positions of the entries to be fetched
           $aux_s$: secret auxiliary information available to the client containing (Count, $sk_{\mathcal{E}}$)
           $\langle A\rangle$: outsourced encrypted array with $n$ entries
    Output: set of entries $\{A[\texttt{first}],\dots,A[\texttt{last}]\}$
1   begin
2       $a \leftarrow$ last $-$ first
3       $\langle h\rangle \leftarrow$ AGGREGATE-PIR-TRAPDOOR($\lfloor \frac{\texttt{first}}{a} \rfloor, a$)
4       ctx $\leftarrow$ AGGREGATE-PIR-SEARCH($\langle h\rangle, \langle Suf\rangle, a$) // executed at server side
5       Chunk$_1$ $\leftarrow$ AGGREGATE-PIR-RETRIEVE(ctx, $a$)
6       $\langle h\rangle \leftarrow$ AGGREGATE-PIR-TRAPDOOR($\lfloor \frac{\texttt{last}}{a} \rfloor, a$)
7       ctx $\leftarrow$ AGGREGATE-PIR-SEARCH($\langle h\rangle, \langle Suf\rangle, a$) // executed at server side
8       Chunk$_2$ $\leftarrow$ AGGREGATE-PIR-RETRIEVE(ctx, $a$)
9       if $\lfloor \frac{\texttt{first}}{a} \rfloor = \lfloor \frac{\texttt{last}}{a} \rfloor$ then
10          return $\{\mathcal{E}$.DEC($sk_{\mathcal{E}}$, Chunk$_2[0]$), ..., $\mathcal{E}$.DEC($sk_{\mathcal{E}}$, Chunk$_2[a-1]$)$\}$
11      else
12          return $\{\mathcal{E}$.DEC($sk_{\mathcal{E}}$, Chunk$_1$[first mod $a$]), ..., $\mathcal{E}$.DEC($sk_{\mathcal{E}}$, Chunk$_1[a-1]$)$\} \cup$
              $\{\mathcal{E}$.DEC($sk_{\mathcal{E}}$, Chunk$_2[0]$), ..., $\mathcal{E}$.DEC($sk_{\mathcal{E}}$, Chunk$_2$[last mod $a$])$\}$
```

The second phase (line 7), corresponding to lines 6–8 in Alg. 1.2, allows to compute the set of positions, in the remotely stored documents, where the leading characters of the occurrences of q are found. Specifically, this set is found in the $o_q = \beta - \alpha$ consecutive entries $Suf[\alpha,\dots,\beta{-}1]$ of the outsourced SA Suf. To retrieve these entries, the client employs the BATCHEDRETRIEVAL procedure, reported in Alg. 3.4, which, given two integers $\texttt{first}$, $\texttt{last}$ and a remotely stored encrypted array $\langle A \rangle$ with n entries, allows to privately retrieve the set of consecutive entries of A from the $\texttt{first}+1$-th one to the $\texttt{last}+1$-th one, i.e., the sub-array $A[\texttt{first}, \ldots, \texttt{last}]$. Specifically, Alg. 3.4 chooses the aggregation factor $a = \texttt{last}-\texttt{first}$ (line 2) and employs two distinct aggregate PIR queries (lines 3–8) to retrieve all the entries in the $\lfloor \frac{\texttt{first}}{a} \rfloor$-th and $\lfloor \frac{\texttt{last}}{a} \rfloor$-th chunk, respectively, among the $\lceil \frac{n}{a} \rceil$ chunks of $\langle A \rangle$. Among the $2a$ entries found in these chunks, the algorithm chooses and decrypts the ones corresponding to the requested entries $\{A[\texttt{first}], \ldots, A[\texttt{last}]\}$ (lines 9–12).

We note that it would be possible to retrieve all the entries with a single PIR aggregate query employing an aggregation factor $a \geq \texttt{last}-\texttt{first}$ chosen as the minimum value such that $\lfloor \frac{\texttt{first}}{a} \rfloor = \lfloor \frac{\texttt{last}}{a} \rfloor$: indeed, this choice guarantees that all the $\texttt{last}-\texttt{first}$ elements $\langle A \rangle[\texttt{first}], \ldots, \langle A \rangle[\texttt{last}]$ are found in the $\lfloor \frac{\texttt{first}}{a} \rfloor$-th chunk among the $\lceil \frac{n}{a} \rceil$ ones of $\langle A \rangle$. Nonetheless, in case the adversary already knows the exact number of occurrences $o_q = \beta - \alpha = \texttt{last} - \texttt{first} + 1$ of the searched string q (e.g., by relying on information related to the application domain), the adversary would learn that $0 \leq \alpha \bmod a \leq a - o_q$, which may in principle allow the adversary to infer meaningful information about the string q. In our PPSS protocol, we thus avoid this optimization to retain a minimum amount of leakage even in case the adversary already knows the number of occurrences of the searched string; nonetheless, this optimization, which allows to save one aggregate PIR query, may be employed in application scenarios where the adversary has no way to know the exact number of occurrences of the searched string, or where the aforementioned information leakage is deemed as acceptable.

The computational and communication costs of the QUERY procedure in our PPSS protocol obviously depend on the corresponding costs of the PIR protocol being employed for the queries: indeed, such costs are mostly dominated by the costs to perform m PIR queries and 2 aggregate PIR queries retrieving o_q consecutive entries. In this work, since we are interested in reducing as much as possible the bandwidth of our solution, we choose to employ the Lipmaa's PIR, because of its polylogarithmic communication cost. By instantiating our construction with Lipmaa's PIR, we achieve an $\mathrm{O}(m \cdot \log(N) b \log_b^2(n) + o_q \log(N) \log_b(\frac{n}{o_q}))$ communication cost for the QUERY procedure, where N is the modulus employed in the DJ LFAHE keypair and b is the radix employed to represent in positional notation an integer index in the Lipmaa's PIR protocol; the computational cost is $\mathrm{O}(m \cdot b \log^3(N) \log_b^4(n) + o_q \cdot \log^2(N) \log_b^5(\frac{n}{o_q}))$ at client side and $\mathrm{O}(m \cdot \frac{n}{b} \log^3(N))$ at server side. Since any PIR protocol can be employed to instantiate our construction of a PPSS protocol, we remark that any improvement in terms of computational or communication costs of a PIR solution may immediately yield a PPSS protocol with better actual performance metrics than the ones reported in this work.

Multi-User Extension

Differently from many of the existing PPSS protocols, our approach can be promptly and efficiently adapted to a multi-user scenario. In this setting, a data-owner outsources the full-text index for a document collection $\mathbf{D}$ to a cloud provider and authorizes multiple users to issue substring search queries over $\mathbf{D}$ to the cloud provider. Our solution allows users to perform queries simultaneously without any synchronization among them and without the need to interact with the data owner.

From an operational point of view the data owner runs the SETUP procedure shown in Alg. 3.1, computing the pair of encrypted arrays $\langle \mathbf{D}\rangle=(\langle A_P\rangle, \langle Suf\rangle)$ to be outsourced and shares the secret auxiliary information $aux_s = (\texttt{Count}, sk_{\mathcal{E}})$ with all authorized users. Then, each authorized user can independently perform substring search queries through the QUERY procedure shown in Alg. 3.2 to find occurrences of a substring of her/his choice: indeed, as a PIR protocol does not modify the data structure stored at server side, PIR queries can be simultaneously performed with no synchronization issues.

Nonetheless, when a naive implementation of Lipmaa's PIR is employed, the scalability of our solution is limited by the memory footprint of PIR queries, as each of them requires an $\mathrm{O}(n)$ additional memory. Indeed, each run of the PIR-SEARCH procedure in Lipmaa's protocol performs $t = \lceil \log_b(n)\rceil$ iterations, with the i-th iteration computing an array $\langle A_{h_{i-1}}\rangle$ with $\lceil \frac{n}{b^i}\rceil$ elements. In particular, the first iteration computes an array $\langle A_{h_0}\rangle$ with $\lceil \frac{n}{b}\rceil$ entries, in turn requiring $\mathrm{O}(n)$ memory to be allocated. Therefore, if u queries are performed simultaneously, the memory consumption of Lipmaa's protocol is $\mathrm{O}(n + u \cdot n)$, in turn easily leading to memory exhaustion in case of multiple queries. To address this scalability issue, we propose to employ a modified PIR-SEARCH procedure, reported in Alg. 3.5, which aims at reducing the additional memory required by each PIR query. This goal is achieved in our optimized

Algorithm 3.5: Optimized PIR-SEARCH algorithm of Lipmaa's PIR

```
Function PIR-SEARCH($A$, $\langle h\rangle$):
    Input: $A$, remote array with $n$ entries;
           $\langle h\rangle$, privacy-preserving representation of the position $h \in \{0,\dots,n-1\}$, represented as the
           concatenation of the bit-wise encryptions of each $b$-bit string in the sequence
           $\texttt{hdigit}_0,\dots,\texttt{hdigit}_{t-1}$, with $t = \lceil \log_b(n)\rceil$ (see Sec. 1.5)
    Output: content of the cell $A[h]$
    Data: $b \geq 2$, radix chosen by the client to construct $\langle h\rangle$
    begin
        return RECURSIVERET ($\langle h\rangle$, $A$, $t-1$, 0, $b^t-1$, $b$, $n$)

Function RECURSIVERET($\langle h\rangle$, $A$, $l$, begin, end, $b$, $n$):
    if begin $\geq n$ then
        return 1 // Stop recursion to avoid overflows
    if end $-$ begin $= 0$ then
        return $A[\texttt{begin}]$
    size $\leftarrow \lfloor \frac{\texttt{end}-\texttt{begin}}{b} \rfloor$, acc $\leftarrow 1$
    for $i \leftarrow 0$ to $b-1$ do
        el $\leftarrow$ RECURSIVERET($\langle h\rangle, A, l-1, \texttt{begin}, \texttt{begin} + \texttt{size}, b, n$)
        begin $\leftarrow$ begin $+$ size $+ 1$
        acc $\leftarrow (\texttt{acc} \cdot \langle h\rangle[(l{\cdot}b{+}i]^{\texttt{el}}) \bmod N^{l+1}$
    return acc
```

PIR-SEARCH procedure by employing a different schedule of its operations. Specifically, the naive PIR-SEARCH procedure serializes the computation of the entire arrays $\langle A_{h_0}\rangle,\dots,\langle A_{h_{t-1}}\rangle$. Nonetheless, it is possible to compute the element $\langle A_{h_1}\rangle[0]$ as soon as the b elements $\langle A_{h_0}\rangle[0],\dots,\langle A_{h_0}\rangle[b-1]$ are computed, and, similarly, compute $\langle A_{h_1}\rangle[1]$ as soon as the b elements $\langle A_{h_0}\rangle[b],\dots,\langle A_{h_0}\rangle[2b-1]$ are computed. In general, the element $\langle A_{h_i}\rangle[j]$, $1{\leq}i{\leq}t-1$, $0{\leq}j{<}\lceil\frac{n}{b^{i+1}}\rceil$, is ready to be computed as soon as the

b elements $\langle A_{h_{i-1}} \rangle[b{\cdot}j], \ldots, \langle A_{h_{i-1}} \rangle[b{\cdot}j{+}b{-}1]$ are available. We note that an element of $\langle A_{h_0} \rangle$ is always ready to be computed as all the elements in the dataset A are available; therefore, to avoid that all elements of $\langle A_{h_0} \rangle$ are computed earlier than all the other elements of arrays $\langle A_{h_1} \rangle, \ldots, \langle A_{h_{t-1}} \rangle$, we rely on the following rule to schedule the operations: an entry of an array $\langle A_{h_i} \rangle$, $0{\le}i{\le}t{-}1$, is computed only if there are no entries of arrays $\langle A_{h_j} \rangle$, $t{-}1{\ge}j{>}i$, that are ready to be computed. This schedule of the operations is achieved by the recursive computation in Alg. 3.5.

The computational complexity of this algorithm is clearly equivalent to the naive iterative implementation, as the same operations are performed. Nonetheless, it exhibits a sublinear memory consumption per query. Indeed, the maximum depth of recursion is $\mathrm{O}(\log(n))$, which means that only the memory for the $\mathrm{O}(\log(n))$ recursive calls is required. Each recursive call stores $\mathrm{O}(l \log(N))$ bits due to DJ LFAHE ciphertexts in $\mathbb{Z}^*_{N^{l+1}}$, thus the overall storage cost is: $\sum_{l=1}^{\lceil \log(n) \rceil} \mathrm{O}(l \log(N)) {=} \mathrm{O}(\log^2(n) \log(N))$. In conclusion, when u queries are simultaneously performed, the server needs only $\mathrm{O}(n{+}u{\cdot}\log^2(n))$ memory, with significant savings w.r.t. a naive approach.

Concerning the security guarantees of our multi-user solution, we note that each user can employ its own keypair for the DJ LFAHE scheme in order to perform PIR queries. This guarantees that each user can perform its own substring search query without leaking any information to both other users and the service provider itself. Indeed, the search and access pattern privacy of the queries of a user are guaranteed even in case of collusion between other users and the service provider, as a colluding user observing queries of other users have no more information than the server about the query, thanks to the adoption of independent LFAHE keypairs among users.

Verifiability of retrieved data

In the following, we enhance the design of our PPSS protocol by providing a simple, yet effective, mechanism that allows clients to verify the correctness of the retrieved data with strong guarantees that they have not been accidentally tampered with by the storage service provider. In the considered semi-honest adversarial settings, the storage service provider is trustworthy to execute the steps of the PPSS protocol faithfully, even if it is interested to acquire as much information as possible on the stored data. However, chances of accidental or misconfiguration errors in the implementation or deployment of the protocol at server side make a mechanism to verify the correctness of accessed data elements a desirable feature.

We introduce a mechanism allowing the client to check throughout a query whether the retrieved element matches the one prepared by the data owner or not. In particular, the retrieved entry may either be corrupted or corresponding to an entry different from the requested one. To this end, to detect that an outsourced data element is corrupted, a cryptographic (keyed) Message Authentication Code (MAC) is employed, while to prevent the chance that another intact entry is retrieved in place of the requested one, each entry of the array is associated with a unique MAC key, paired with the entry at hand. Specifically, the value of the secret cryptographic key employed to compute the MAC of each entry of the outsourced array must fulfill the following properties: *i)* it must depend on the index of the array entry; *ii)* it must be efficiently computable by the client given the index of an entry as an input.

In our design both properties are provided by generating the MAC key of an array entry via a keyed Pseudo Random Function (PRF), which is fed with a (secret) master key, shared by the data owner with each user, and the index of the entry at hand. A cryptographic PRF is an efficiently-computable function which emulates a random oracle. In particular, there is no efficient algorithm able to distinguish (with significant advantage) between the output of the chosen PRF and the output of a random oracle, i.e., the outputs of a PRF appear indistinguishable from random payloads by the adversary.

Our PPSS protocol is thus extended in the following way. The data owner selects a master secret key $\texttt{msk} \xleftarrow{\mathcal{R}} \{0,1\}^\lambda$, where λ is the bit-length of the randomly chosen value (e.g., $\lambda \in \{128, 192, 256\}$), and a keyed PRF $F : \{0,1\}^\lambda \times \{0,1\}^{\lceil \log(n) \rceil} \mapsto \{0,1\}^\lambda$, which takes as input a master key and an integer value denoting the index of an n-cell array. A possible instance of the said PRF is given by the AES-CBC encryption function yielding only the last block of the ciphertext and employing a λ-bit key, $\lambda \in \{128, 192, 256\}$. Subsequently, the value of each entry of the array A to be outsourced is augmented with its corresponding MAC: that is, the i-th entry, $i \in \{0, \dots, n-1\}$, of the array A, which contains the value a_i, is replaced with the pair of values $(a_i, \text{MAC}(k_i, a_i))$, where $k_i = F(\texttt{msk}, i)$. When the client issues a query to retrieve the array element with index h, it gets back the pair of values $(a_{h'}, \text{MAC}(k_{h'}, a_{h'}))$ and can verify the correctness of the retrieved data by deriving $k_h = F(\texttt{msk}, h)$ and checking if the re-computation of $\text{MAC}(k_h, a_{h'})$ yields the same value $\text{MAC}(k_{h'}, a_{h'})$ returned by the server, thus concluding whether $h = h'$ or not. In case of an unsuccessful verification, the client can provide strong (cryptographic) evidence that the server either returned an accidentally corrupted element or wrongly returned an intact entry in place of the requested one. We remark that the communication and computational overheads due to the transmission of the MAC value and its re-computation at client-side impact on the overall performance of the PPSS protocol in a negligible way. Indeed, they involve the transmission of a few tens of bytes and the computation of a symmetric-key cryptographic primitive, which is way more efficient (by two or three orders of magnitude) than the asymmetric cryptographic operations performed at client side throughout the Lipmaa's PIR protocol.

3.2 Queries with Wildcard Characters

We now extend our PPSS protocol to enable queries for a string q containing meta-characters, also called wildcards, that allow to define a language (i.e., a set of strings) over the alphabet Σ instead of a single string. We call a pattern, denoted by $\mathfrak{p}$ from now on, any string containing at least one of these wildcards; the language defined by a pattern $\mathfrak{p}$ is denoted by $L_\mathfrak{p}$. Although a pattern $\mathfrak{p}$ is usually employed to filter out strings that do not belong to $L_\mathfrak{p}$, in our PPSS protocol we want to find the positions of the (sub)strings in the document collection $\mathbf{D}$ that are also in $L_\mathfrak{p}$. Henceforth, these (sub)strings will be referred to as *matches* or *occurrences* of the pattern $\mathfrak{p}$ over the document collection; each occurrence is identified by the document where it is located and its starting position in the document at hand:

Definition 3.1 (Occurrence of a Pattern). *Given a document collection* $\mathbf{D}$ *with* $z \geq 1$ *documents* $\{D_1, \dots, D_z\}$ *and a pattern* $\mathfrak{p}$*, the set of positions of the occurrences of* $\mathfrak{p}$ *in* D_i*,* $i \in \{1, \dots, z\}$*, is defined as:*

$$O_{D_i,\mathfrak{p}} = \{\, j \mid 0 \leq j \leq \text{LEN}(D_i) - 1 \wedge \exists k\, (D_i[j, \dots, k] \in L_\mathfrak{p},\, \text{LEN}(D_i) - 1 \geq k \geq j)\}$$

Table 3.1: *Wildcard meta-characters available in the Unix glob patterns with their semantic*

Wildcard	Semantic
$*$	Match zero or more arbitrary characters
$?$	Match exactly one character
$[abc]$	Match either a, b or c
$[0-9]$	Match a single digit
$[!ACG]$	Match any character except for A, C ad G
$[!A-Z]$	Match any character except for uppercase letters

Coherently with Def. 1.5, our PPSS protocol aims at computing, for a pattern $\mathfrak{p}$ and a document collection $\mathbf{D}$, the set of positions $O_{\mathbf{D},\mathfrak{p}} = \bigcup_{i=1}^{z} O_{D_i,\mathfrak{p}}$. To specify a pattern $\mathfrak{p}$ in the queries of our PPSS protocol, we define our own format, building upon the well-known *glob patterns*[1], which denote a largely used simple syntax in the Command Line Interface (CLI) of Unix-based systems to filter out the filenames not belonging to the language defined by the specified pattern.

Format of Patterns in Queries

Table 3.1 reports the meta-characters defined in glob patterns and their semantic. Given the set of characters defined by an alphabet Σ, the wildcard $*$ is used to match any sequence of characters (even the empty one) over Σ, while the wildcard $?$ matches exactly one character in Σ. A *character class* denotes a syntax to match exactly one character belonging to a collection (class) of characters specified within square brackets. The collection of characters can be denoted by either listing all of them or specifying the first and the last of them (in alphabetical order) separated by a dash (e.g, $[a-z]$ denotes the class of lowercase latin letters). The meta-character ! can be employed only next to the left square bracket of a character class to "complement" the class at hand, that is matching the elements of Σ different from the ones belonging to the collection of characters specified afterwards (e.g., $[!a-z]$ denotes all the characters in the alphabet except for lowercase latin ones).

To increase the expressiveness of our patterns with respect to glob ones, in our own format we employ the additional meta-character $|$ to denote the *union operator*, i.e., given $k \geq 2$ strings $\beta_1, \ldots, \beta_k$, the pattern $\beta_1|\ldots|\beta_{\mathrm{k}}$ matches any string among the k given ones. Together with the union operator we also introduce *round brackets* as meta-characters to specify unambiguously its scope, e.g., $a(a|b)c$. Besides querying strings containing wildcard characters, we consider prefix and suffix queries, which are quite common in information retrieval contexts [58]. Specifically, a prefix (resp. suffix) query matches only strings that are a prefix (resp. suffix) of each document in the collection $\mathbf{D}$. For instance, the widely used glob pattern $\mathfrak{p}_q = q*$ (resp. $\mathfrak{p}_q = *q$) can be used to issue a prefix (resp. suffix) query, requesting to match all documents starting (resp. ending) with a string equal to q. Nonetheless, since we define an occurrence of a pattern $\mathfrak{p}$ as the position in the document where a sequence of characters match (see Def. 3.1), we need to introduce a special symbol to specify that such a sequence must appear at the beginning or at the end of the document. To this extent, we add to our format the special symbol $\&$, called *meta-delimiter*, which should appear only as the

[1]http://man7.org/linux/man-pages/man3/glob.3.html

first or last character of a pattern.

The definition of the format of patterns in queries of our PPSS protocol is formally captured by Def. 3.4, with Definitions 3.2, 3.3 being introduced to properly frame the use of the wildcard $*$.

Definition 3.2 (Star-Free Pattern). *Given an alphabet Σ, the set $\mathcal{G}$ of glob wildcards reported in Tab. 3.1, the union operator $|$ and the round brackets meta-characters, a pattern $\mathfrak{p}$ is* star-free *if and only if it is built as the concatenation of $k{\geq}1$ strings $\mathfrak{p} = \alpha_1\alpha_2\cdots\alpha_k$, where each α_i belongs to one of the following types:*

1. *$\alpha_i \in \Sigma^+$ (strings with at least one character composed only by characters in Σ)*
2. *$\alpha_i \in \{\mathcal{G} \setminus \{*\}\}^+$ (strings with at least one meta-character composed only by meta-characters in $\mathcal{G}$, except $*$)*
3. *$\exists\, \beta_1, \cdots, \beta_h, h{\geq}2 : \alpha_i = (\beta_1|\ldots|\beta_h) \vee \alpha_i = (\beta_1|\cdots|\beta_{h-1}|\varepsilon)$, where all $\beta_1, \cdots, \beta_h$ are star-free patterns.*
   ε denotes here the empty string (i.e., a string with no characters) that is appended to a meta-character $|$ to point out a possible match with no character in Σ.

Definition 3.3 (Well formed Star-Free Pattern). *Given a star-free pattern $\mathfrak{p} = \alpha_1\cdots\alpha_k$ over the alphabet Σ, it is a* well formed star-free *pattern if and only if there exists $1 \leq h \leq k$ such that $\alpha_h \in \Sigma^+$.*

A query string is considered as a well formed star-free pattern if it contain at least one type (1) substring and does not contain the wildcard $*$. It is worth noting that such a query string may contain the union operator only if it is applied to star-free patterns (not necessarily well formed). We introduce such a restriction for simplicity and efficiency reasons as our privacy-preserving queries for a pattern $\mathfrak{p}$ containing the union operator exhibits a computational complexity linear in the length of the longest string among the ones defined by $\mathfrak{p}$ (i.e., belonging to the language $L_{\mathfrak{p}}$). Thus, the presence of a $*$ wildcard would easily increase the complexity to be linear in the size of the outsourced index.

Definition 3.4 (Well Formed Patterns). *Given an alphabet Σ, the set $\mathcal{G}$ of glob wildcards reported in Tab. 3.1, the union operator $|$, the round brackets meta-characters, and the meta-delimiter symbol $\&$, a pattern $\mathfrak{p}$ is a* well formed pattern *if and only if there exist $k \geq 1$ well formed star-free patterns $\alpha_1, \cdots, \alpha_k$ such that $\mathfrak{p}$ exhibits one of these structures:*

1. *$p = \alpha_1 * \alpha_2 * \cdots * \alpha_k$*
2. *$p = \&\alpha_1 * \alpha_2 * \cdots * \alpha_k$ (prefix pattern)*
3. *$p = \alpha_1 * \alpha_2 * \cdots * \alpha_k\&$ (suffix pattern)*
4. *$p = \&\alpha_1 * \alpha_2 * \cdots * \alpha_k\&$ (prefix-suffix pattern)*

We note that a well formed pattern has some restrictions on the usage of $*$ wildcard: indeed, the structure of a well formed pattern mandates that it cannot start or end with a $*$. We introduce this restriction since a pattern $\mathfrak{p}_\gamma = *\gamma*$, $\gamma \in \Sigma^+$, in our format would match any sequence of characters in the document collection having γ as a substring thus yielding too many (unuseful) occurrences (see Def. 3.1). Indeed, given a document D_i, $i \in \{1, \ldots, z\}$, $z \geq 1$, in the collection $\mathbf{D}$, if j is the position of an occurrence of $\mathfrak{p}_\gamma$ in D_i, then every position $0 \leq h \leq j$ is an occurrence of $\mathfrak{p}_\gamma$ in D_i.

PPSS Protocol for Well-Formed Patterns

In the following, we show how to extend our PPSS protocol to deal with queries asking for the occurrences of a well-formed pattern $\mathfrak{p}$. We proceed in two steps: we first show how to perform privacy-preserving queries asking for the occurrences of a well-formed star-free pattern (see Def. 3.3); then, we show how to perform queries for a generic well-formed pattern (see Def. 3.4), relying on the query algorithm for well-formed star-free patterns as a building block.

Queries for Well-formed Star-free Patterns. Our PPSS protocol performs privacy-preserving queries for well-formed star-free patterns by hinging upon the following decomposition of the input pattern.

Lemma 3.1 (Decomposition of Well-Formed Star-Free Pattern). *Given a well-formed star-free pattern $\mathfrak{p}$ over an alphabet Σ, there exists a set of $2k + 1$, $k \geq 1$, strings $\{\gamma_0, \omega_1, \gamma_1, \ldots, \omega_k, \gamma_k\}$ such that $\mathfrak{p} = \gamma_0 \omega_1 \gamma_1 \ldots \omega_k \gamma_k$, where:*

- *$\omega_1, \ldots, \omega_k \in \Sigma^+$ (i.e., type (1) strings in Def. 3.3)*
- *$\gamma_0, \ldots, \gamma_k$ are a concatenation of type (2) or type (3) strings only (see Def. 3.3)*
- *γ_0 and γ_k may be equal to the empty string ε*

Proof. Following Def. 3.3, a well-formed star-free pattern $\mathfrak{p}$ is composed by $h \geq 1$ type (1), (2) or (3) strings, $\mathfrak{p} = \alpha_1 \cdots \alpha_h$, where at least one of them is a type (1) string. We prove the lemma by induction over the number h of type (1), (2) or (3) strings composing $\mathfrak{p}$. Assuming $h = 1$, there is only the pattern $\mathfrak{p} = \alpha_1$, where α_1 is a type (1) string: in this case, $\mathfrak{p}$ can be decomposed as $\gamma_0 \alpha_1 \gamma_1$, where $\gamma_0 = \gamma_1 = \varepsilon$, which satisfies the lemma. Assuming $h = 2$, $\mathfrak{p} = \alpha_1 \alpha_2$, there are two possible cases: if both α_1 and α_2 are type (1) strings, then $\mathfrak{p}$ can be decomposed as $\gamma_0 \omega_1 \gamma_1$, where $\gamma_0 = \gamma_1 = \varepsilon$ and $\omega_1 = \alpha_1 \alpha_2$, which satisfies the lemma; if α_1 (resp. α_2) is a type (1) string and α_2 (resp. α_1) is either a type (2) or a type (3) string, then $\mathfrak{p}$ can be decomposed as $\gamma_0 \alpha_1 \alpha_2$ (resp. $\alpha_1 \alpha_2 \gamma_1$), where $\gamma_0 = \varepsilon$ (resp. $\gamma_1 = \varepsilon$), which satisfies the lemma. We now proceed with the inductive step: we want to prove that any well-formed star-free pattern $\mathfrak{p}$ composed by $h{+}1$ type (1), (2) or (3) strings $\mathfrak{p} = \alpha_1 \cdots \alpha_{h+1}$ can be decomposed as in the lemma, assuming that any well-formed star-free pattern composed by h type (1), (2) or (3) strings can be decomposed as in the lemma. Consider $\mathfrak{p} = \alpha_1 \mathfrak{p}'$, with $\mathfrak{p}' = \alpha_2 \cdots \alpha_{h+1}$. The pattern $\mathfrak{p}'$ is composed by h strings, thus, by inductive hypothesis, it can be decomposed as $\gamma'_0 \omega'_1 \gamma'_1 \cdots \omega'_k \gamma'_k$ for a $k \geq 1$. To encompass a generic pattern $\mathfrak{p}$ with $h{+}1$ strings, we have 4 different cases to consider:

1. α_1 is a type (1) string and $\gamma'_0 = \varepsilon$, as a consequence $\mathfrak{p}$ can be decomposed as $\gamma'_0 \omega_1 \gamma'_1 \cdots \omega'_k \gamma'_k$, with $\omega_1 = \alpha_1 \omega'_1$
2. α_1 is a type (1) string and $\gamma'_0 \neq \varepsilon$, as a consequence $\mathfrak{p}$ can be decomposed as $\gamma_0 \alpha_1 \gamma'_0 \omega'_1 \gamma'_1 \ldots \omega'_k \gamma'_k$, with $\gamma_0 = \varepsilon$

3. α_1 is either a type (2) or type (3) string and $\gamma'_0 = \varepsilon$, as a consequence $\mathfrak{p}$ can be decomposed as $\alpha_1\omega'_1\gamma'_1 \dots \omega'_k\gamma'_k$
4. α_1 is either a type (2) or type (3) string and $\gamma'_0 \neq \varepsilon$, as a consequence $\mathfrak{p} = \gamma_0\omega'_1\gamma'_1 \dots \omega'_k\gamma'_k$, with $\gamma_0 = \alpha_1\gamma'_0$

□

In our PPSS protocol, an occurrence is identified by a pair (pos, off) denoting, respectively, the starting position, in the string $s = D_1\$D_2\$ \dots D_z\$$ derived from the outsourced document collection $\mathbf{D} = \{D_1, \dots, D_z\}$, of the document D_i where the occurrence is located (pos), and the relative position of the occurrence in D_i (off). Along with this information, STARFREEQUERY, our privacy-preserving query procedure for well-formed star-free patterns reported in Alg. 3.6, associates to each of the occurrences in the returned set $R_\mathfrak{p}$ also its ending position in the document D_i where it is located, as this information is needed in the query procedure for a well-formed pattern, which employs STARFREEQUERY as a building block. Analogously to the QUERY procedure reported in Alg. 3.2, the STARFREEQUERY procedure in Alg. 3.6 takes as input a triple consisting of a well-formed star-free pattern $\mathfrak{p}$, the auxiliary secret information which is needed to decrypt the elements retrieved from outsourced data structures, and the privacy-preserving representation of the document collection $\langle\mathbf{D}\rangle$. Nonetheless, $\langle\mathbf{D}\rangle$ has to be enriched with the encrypted string $\langle s\rangle$, computed through the same semantically secure cipher $\mathcal{E}$ and the same keypair $(pk_\mathcal{E}, sk_\mathcal{E})$ employed for the original components of the privacy preserving representation of the document collection, i.e., $\langle A_P\rangle$ and $\langle \mathit{Suf}\rangle$. We note that the encrypted string $\langle s\rangle$ can be stored in place of the encrypted document collection $\mathbf{D}$, hereby avoiding an additional storage overhead.

The STARFREEQUERY procedure follows the decomposition of the well-formed star-free pattern $\mathfrak{p}$ defined in Lemma 3.1, parsing it properly (line 2). If the pattern is composed by a single type (1) string (line 3), the algorithm invokes the QUERY procedure in Alg. 3.2, as shown in line 4 of Alg. 3.6. After getting the occurrences of the pattern, the STARFREEQUERY procedure enriches each of them with the corresponding ending position in the document where they are located. Such an ending position is computed considering the starting position and the length of the instance of the pattern at hand as shown in lines 5–7.

In case the pattern $\mathfrak{p}$ has some wildcards, the algorithm proceeds in two phases. In the first one (lines 8–13), the client considers the $k{\geq}1$ type (1) strings $\omega_1, \dots, \omega_k$ in $\mathfrak{p}$, with the aim of locating the one with the minimum number of occurrences in $\mathbf{D}$. To this extent, the algorithm executes, for each type (1) string ω_i, $i \in \{1, \dots, k\}$, the same steps performed in the `Qnum` phase of the QUERY procedure in Alg. 3.2 (denoted by the QUERYNUM procedure in line 10), computing the indexes α, β identifying the portion $\mathit{Suf}[\alpha, \dots, \beta{-}1]$ of the SA that stores the $o_{\omega_i} = \beta - \alpha$ positions of the occurrences of ω_i; at the end of this loop, the variables $\alpha_{min}, \beta_{min}$ identify the portion of the SA corresponding to the occurrences of the string ω_{min} with the least number of occurrences among ω_i ones, $i \in \{1, \dots, k\}$ (lines 11–12). Then, the algorithm employs the BATCHEDRETRIEVAL procedure reported in Alg. 3.4 to retrieve the set `Occ` of occurrences of ω_{min} with two aggregate PIR queries (line 13).

The second phase (lines 14–25) uses the occurrences in `Occ` to finally compute the ones of the pattern $\mathfrak{p}$, returning them in the output set $R_\mathfrak{p}$. Indeed, each sequence of

Algorithm 3.6: Query procedure for well-formed star-free patterns in our PPSS protocol

```
Function StarFreeQuery($\mathfrak{p}$, $aux_s$, $\langle\mathbf{D}\rangle$):
    Input: $\mathfrak{p}$: well-formed star-free pattern to be searched
           $aux_s = (\texttt{Count}, sk_{\mathcal{E}})$: secret auxiliary information employed by the client for the queries
           $\langle\mathbf{D}\rangle = (\langle A_P\rangle, \langle Suf\rangle, \langle s\rangle)$: remotely accessed privacy-preserving representation of $\mathbf{D}$
    Output: $R_{\mathfrak{p}}$: set of starting and ending positions of the occurrences of $\mathfrak{p}$ in $\mathbf{D}$
    begin
        $(\gamma_0, \omega_1, \ldots, \omega_k, \gamma_k) \leftarrow$ ParsePattern($\mathfrak{p}$)
        if $k = 1 \wedge \gamma_0 = \varepsilon \wedge \gamma_k = \varepsilon$ then
            $\texttt{Occ} \leftarrow$ Query($\omega_1$), $R_{\mathfrak{p}} \leftarrow \emptyset$
            foreach $\texttt{o} \in \texttt{Occ}$ do
                /* Alg. 3.2 returns, for each occurrence o, the pair
                $(pos, off)$ denoting the starting position in $s$ of the document
                where o is located and its relative position from it */
                $R_{\mathfrak{p}} \leftarrow R_{\mathfrak{p}} \cup (\texttt{o}.pos, \texttt{o}.off, \texttt{o}.off + \text{LEN}(\omega_1) - 1)$
            return $R_{\mathfrak{p}}$
        $\alpha_{min} \leftarrow 0, \beta_{min} \leftarrow n$
        for $i \leftarrow 1$ to $k$ do
            $(\alpha, \beta) \leftarrow$ QueryNum($\omega_i$, $aux_s$, $\langle A_P\rangle$)
            if $\beta - \alpha < \beta_{min} - \alpha_{min}$ then
                $\beta_{min} \leftarrow \beta, \alpha_{min} \leftarrow \alpha, \omega_{min} \leftarrow \omega_i$
        $\texttt{Occ} \leftarrow$ BatchedRetrieval($\alpha_{min}, \beta_{min} - 1, aux_s, \langle Suf\rangle$)
        $R_{\mathfrak{p}} \leftarrow \emptyset, \texttt{len} \leftarrow$ ComputeMaxLength($\omega_{min} \ldots \omega_k\gamma_k$)
        $\texttt{max\_len} \leftarrow$ ComputeMaxLength($\mathfrak{p}$), $\texttt{min\_len} \leftarrow$ ComputeMinLength($\mathfrak{p}$)
        foreach $o \in \texttt{Occ}$ do
            $\texttt{end} \leftarrow o.pos + o.off + \texttt{len} - 1$
            $\texttt{start}_1 \leftarrow \texttt{end} - \texttt{max\_len} + 1, \texttt{start}_2 \leftarrow \texttt{end} - \texttt{min\_len} + 1$
            $\texttt{str} \leftarrow$ BatchedRetrieval($\texttt{start}_1, \texttt{end}, \langle s\rangle, aux_s$)
            for $j \leftarrow \texttt{start}_1$ to $\texttt{start}_2$ do
                $\texttt{match\_len} \leftarrow$ MatchShortestPrefix($\texttt{str}[j - \texttt{start}_1, \ldots, \texttt{max\_len} - 1], \mathfrak{p}$)
                if $\texttt{match\_len} > 0$ then
                    $\texttt{offset} \leftarrow j - 1 - o.pos$
                    $R_{\mathfrak{p}} \leftarrow R_{\mathfrak{p}} \cup (o.pos, \texttt{offset}, \texttt{offset} + \texttt{match\_len} - 1)$
        return $R_{\mathfrak{p}}$
```

characters matching the pattern $\mathfrak{p}$ must contain ω_{min} as a substring, thus, for each occurrence of the pattern $\mathfrak{p}$, there must be an occurrence $o \in \texttt{Occ}$ with position $p_o = o.pos + o.off$ over s between the starting and ending positions over s of the occurrence of $\mathfrak{p}$ at hand. As a consequence, the occurrences of $\mathfrak{p}$ are computed by analyzing the characters of s preceding and succeeding each occurrence $o \in \texttt{Occ}$. To this end, Alg. 3.6 employs the two procedures ComputeMaxLength and ComputeMinLength, which, given a star-free pattern $\mathfrak{p}$, compute the lengths of the longest and shortest strings in $L_{\mathfrak{p}}$ (the set of all possible strings matching $\mathfrak{p}$), respectively. Indeed, since a star-free pattern may contain a union operator applied to strings $\beta_1, \ldots, \beta_h$ with different lengths, the length of an occurrence of a pattern may vary depending on which string among $\beta_1, \ldots, \beta_h$ is matched in the occurrence at hand. In the StarFreeQuery procedure, the algorithm first employs the ComputeMaxLength procedure to obtain the length, `len`, of the longest possible string matching the pattern $\mathfrak{p}$ shortened to start from ω_{min} (line 14); then, it computes the lengths of the longest and shortest strings in $L_{\mathfrak{p}}$ as `max_len` and `min_len` at line 15.

For each occurrence $o \in \texttt{Occ}$, the biggest possible position of the final character of a substring matching $\mathfrak{p}$ and including $s[p_o]$, $p_o = o.pos + o.off$, is computed as $\texttt{end} = p_o{+}\texttt{len}{-}1$ (line 17). The range $\{\texttt{start}_1, \cdots, \texttt{start}_2\}$, identifying the possible positions of the leading character of a substring matching $\mathfrak{p}$ and including $s[p_o]$, $p_o = o.pos{+}o.off$, is found by computing $\texttt{start}_1 = \texttt{end}{-}\texttt{max\_len}{+}1$ and $\texttt{start}_2 = \texttt{end}{-}\texttt{min\_len}{+}1$ (line 18). Subsequently, the BATCHEDRETRIEVAL procedure reported in Alg.3.4 is employed to privately retrieve with two aggregate PIR queries the sequence of characters $\texttt{str} = s[\texttt{start}_1, \cdots, \texttt{end}]$ from the encrypted string $\langle s\rangle$ stored at server side (line 19). The retrieved sequence of characters $\texttt{str}$ includes all the occurrences of $\mathfrak{p}$ containing the character $s[p_o]$ and with leading character positioned in the range $\{0, \ldots, \texttt{start}_2{-}\texttt{start}_1\}$ over $\texttt{str}$. The occurrences of $\mathfrak{p}$ over $\texttt{str}$ are thus identified by searching for the shortest prefix of $\texttt{str}[j, \cdots, \texttt{max\_len} - 1]$, $0{\leq}j{\leq}\texttt{start}_2 - \texttt{start}_1$, which is matched by $\mathfrak{p}$ (i.e., the shortest prefix that belongs to $L_\mathfrak{p}$), employing the MATCHSHORTESTPREFIX procedure in line 21. If the said prefix match exists, the starting and ending positions of the occurrence $\mathfrak{p}$ are inserted in the output set $R_\mathfrak{p}$ together with the starting position of the document in s where the occurrence at hand is located (i.e., $o.pos$), (lines 22–24).

We now analyze the computational and communication costs of the STARFREEQUERY algorithm fed with a pattern $\mathfrak{p}$. To this extent, we consider the decomposition of the pattern $\mathfrak{p}$ reported in Lemma 3.1, that is $\mathfrak{p} = \gamma_0\omega_1\gamma_1 \ldots \omega_k\gamma_k$, $k{\geq}1$. In our analysis, we denote as o_{ω_i} the number of occurrences of ω_i in the document collection $\mathbf{D}$, as m the length of the longest possible string matching $\mathfrak{p}$, i.e., m = COMPUTEMAXLENGTH($\mathfrak{p}$), and as $m_\omega = \sum_{i=1}^{k} \textsc{len}(\omega_i)$ the sum of the lengths of type (1) strings $\omega_1, \ldots, \omega_k$.

Concerning the communication cost, in the first phase of the algorithm (lines 2–13) the client sends $\mathrm{O}(m_\omega b \log_b^2(n)\log(N))$ bits to the server (with N and b being parameters employed in the Lipmaa's PIR protocol as the cryptographic modulus of the LFAHE scheme and the radix value to compute trapdoors, respectively), while the server sends back $\mathrm{O}((m_\omega{+}o_{min}){\cdot}\log_b(n)\log(N))$ bits, where $o_{min} = \texttt{min}(o_{\omega_1}, \ldots, o_{\omega_k})$; this cost is largely dominated by the communication cost of the second phase (lines 14–25), which amounts to $\mathrm{O}(o_{min} \cdot (b\log_b^2(n)\log(N) + m\log_b(n)\log(N)))$, that is the cost of o_{min} PIR queries, each of which retrieving $\mathrm{O}(m)$ characters from $\langle s\rangle$.

The computational cost at client side, which amounts to $\mathrm{O}(o_{min}{\cdot}(b\log_b^4(n)\log^3(N){+} m\log_b^5(n)\log^2(N)))$, is mostly due to the o_{min} queries that retrieve $\mathrm{O}(m)$ characters from $\langle s\rangle$; indeed, the computational cost to match the pattern $\mathfrak{p}$ with the string downloaded from the server is negligible with respect to the cryptographic operations. Finally, the computational cost at server side amounts to $\mathrm{O}((m_\omega{+}o_{min}) \cdot \frac{n}{b}\log^3(N))$, which is obtained by adding the cost of the m_ω PIR queries performed over the k invocations of the QUERYNUM procedure and the cost of the o_{min} aggregate PIR queries to retrieve substrings from $\langle s\rangle$.

Dealing with Meta-Delimiters. To employ the STARFREEQUERY algorithm as a building block during the execution of queries with well-formed patterns (Def. 3.4), we need to extended it to manage also the meta-delimiters preceding or succeeding a well-formed star-free pattern $\mathfrak{p}$, which are employed in prefix, suffix and prefix-suffix queries. To this extent, the PARSEPATTERN procedure converts each meta-delimiter character & found in $\mathfrak{p}$ into the end-of-string delimiter \$ employed to concatenate the documents in the collection $\mathbf{D}$ into the string s. Two cases are possible:

- If the delimiter \$ precedes (resp. succeeds to) the type (1) string ω_1 (resp. ω_k), then it is merged with the string at hand and the StarFreeQuery procedure proceeds as shown in Alg. 3.6. Indeed, the type (1) string enriched with the merged symbol \$ may be processed by either the Query or the QueryNum procedure at lines 4 and 10, respectively, as both procedures support prefix, suffix and prefix-suffix queries without any further modifications. Indeed, any occurrence of pattern $\$\omega_1$ (resp. $\omega_k\$$) identified by these algorithms can be positioned only at the beginning (resp. end) of a document in s, while any occurrence of the pattern $\$\omega_1\$$ identified by these algorithms correspond to an entire document being equal to ω_1.

- If the delimiter \$ neither precedes nor succeeds a type (1) string, then the algorithm exploits the fact that the number of occurrences of the pattern with the meta-delimiters is at most z, one for each document D_i in $s = D_1\$ \cdots D_z\$$. Therefore, if, at the end of the loop at lines 9–12, the string ω_{min} has more than z occurrences, then the StarFreeQuery, instead of retrieving its occurrences at line 13, retrieves the occurrences of \$ in s. Specifically, these are stored in the first z entries of the suffix array and so they can be retrieved at line 13 by computing BatchedRetrieval$(0, z{-}1, aux_s, \langle Suf \rangle)$. Obviously, since in this case the occurrences in `Occ` no longer refers to the string ω_{min} but to the symbol \$, the computation of the variable `len` at line 14 must be modified accordingly; specifically, if the meta-delimiter precedes $\mathfrak{p}$, then `len` equals ComputeMaxLength$(\mathfrak{p})$ plus the number of meta-delimiters (which is at most two), otherwise `len` $= 1$.

Queries with Well-formed Patterns. The procedure QueryPattern in Alg. 3.7 allows to locate all the occurrences of a well-formed pattern in our PPSS protocol. The algorithm hinges upon the decomposition of a well-formed pattern $\mathfrak{p}$ in a set of $k{\geq}1$ well-formed star-free patterns $\alpha_1, \dots, \alpha_k$ (see Def. 3.4). Specifically, given all the occurrences of each of these k patterns in the document collection $\mathbf{D} = \{D_1, \dots, D_z\}$, it is possible to construct an occurrence of $\mathfrak{p}$ over a document D_j in $\mathbf{D}$, $1{\leq}j{\leq}z$, by finding a set of occurrences $\mathfrak{o}_{k,j} = \{o_{1,j}, \dots, o_{k,j}\}$ located in D_j such that $o_{i,j}$ is an occurrence of α_i, $i \in \{1, \dots, k\}$, in D_j, and $\forall\, i{\leq}k{-}1$, $o_{i,j}.end < o_{i+1,j}.begin$, where $o_{i,j}.end$ (resp. $o_{i,j}.begin$) denotes the ending (resp. starting) position in D_j of the occurrence $o_{i,j}$. Indeed, $\mathfrak{o}_{k,j}$ identifies an occurrence of $\mathfrak{p}$ in document D_j starting at position $o_{1,j}.begin$ and ending at $o_{k,j}.end$, which is composed by occurrences of well-formed star-free patterns $\alpha_1, \dots, \alpha_k$ interleaved by an arbitrary number of characters (due to presence of the wildcard $*$ in $\mathfrak{p}$).

The QueryPattern procedure reported in Alg. 3.7 starts by decomposing the well-formed pattern $\mathfrak{p}$ in $k{\geq}1$ well-formed star-free patterns $\alpha_1, \dots, \alpha_k$ (line 2). Any meta-delimiter $\&$ found at the beginning (resp. at the end) of $\mathfrak{p}$ is merged with the pattern α_1 (resp. α_k). Then, for each of these k patterns, the StarFreeQuery procedure is run to retrieve the set $\mathtt{Occ}_i$ of occurrences of the pattern α_i in the document collection (line 4); each set $\mathtt{Occ}_i$ is further partitioned in z portions ($\mathtt{Occ}_i^1, \dots, \mathtt{Occ}_i^z$), each containing all the occurrences located in the j-th document D_j, $j \in \{1, \dots, z\}$ (line 5). This partitioning is simply performed by grouping the occurrences in $\mathtt{Occ}_i$, $i \in \{1, \dots, k\}$, according to the position pos of the document where each occurrence is located.

Subsequently, for each document D_j, the algorithm (lines 6–9) constructs occurrences of the well-formed pattern $\mathfrak{p}$ from the sets $\mathtt{Occ}_1^j, \dots, \mathtt{Occ}_k^j$, which store the oc-

Algorithm 3.7: Query procedure for well-formed patterns in our PPSS protocol

```
Function QueryPattern($\mathfrak{p}$, $aux_s$, $\langle\mathbf{D}\rangle$):
    Input: $\mathfrak{p}$: well-formed pattern to be searched
           $aux_s = (\mathtt{Count}, sk_{\mathcal{E}})$: secret auxiliary information employed by the client for queries
           $\langle\mathbf{D}\rangle = (\langle A_P\rangle, \langle Suf\rangle, \langle s\rangle)$: remotely accessed privacy-preserving representation of $\mathbf{D}$
    Output: $R_{\mathfrak{p}}$: set of positions of occurrences of $\mathfrak{p}$ in $\mathbf{D}$
1   begin
2       $(\alpha_1, \dots, \alpha_k) \leftarrow$ ParsePattern($\mathfrak{p}$)
3       for $i \leftarrow 1$ to $k$ do
4           $\mathtt{Occ}_i \leftarrow$ StarFreeQuery($\alpha_i, aux_s, \langle\mathbf{D}\rangle$)
5           $(\mathtt{Occ}_i^1, \dots, \mathtt{Occ}_i^z) \leftarrow$ SplitByDocID($\mathtt{Occ}_i$)
6       $R_{\mathfrak{p}} \leftarrow \emptyset$
7       for $j \leftarrow 1$ to $z$ do
8           $R_{\mathfrak{p}} \leftarrow R_{\mathfrak{p}} \cup$ MatchOcc($\mathtt{Occ}_1^j, \dots, \mathtt{Occ}_k^j$)
9       return $R_{\mathfrak{p}}$
```

currences of the corresponding well-formed star-free patterns $\alpha_1, \dots, \alpha_k$ over D_j. The occurrences of $\mathfrak{p}$ are found by the procedure MatchOcc, which, for each $o_{1,j}^h \in \mathtt{Occ}_1^j$, $1 \le h \le |\mathtt{Occ}_1^j|$, computes a set of occurrences $\mathfrak{o}_{k,j}^h$, each corresponding to an occurrence of $\mathfrak{p}$; indeed, although many sets $\mathfrak{o}_{k,j}^h$ may exists for each $o_{1,j}^h$, according to Def. 3.1 they represent the same occurrence of the pattern $\mathfrak{p}$, as they share the same starting position $o_{1,j}^h.begin$. Therefore, there are at most $|\mathtt{Occ}_1^j|$ occurrences of $\mathfrak{p}$ in the document D_j, one for each $o_{1,j}^h$. The MatchOcc procedure can naively construct all these $|\mathtt{Occ}_1^j|$ sets $\mathfrak{o}_{k,j}^h$ in time $O(|\mathtt{Occ}_1^j| \cdot o_{\alpha,j})$, where $o_{\alpha,j} = \sum_{i=1}^{k} |\mathtt{Occ}_i^j|$, that is the sum of the occurrences of each pattern α_i over document D_j; we remark that as $|\mathtt{Occ}_1^j| = O(o_{\alpha,j})$, then the cost of this naive implementation becomes $O(o_{\alpha,j}^2)$.

Algorithm 3.8: Optimized MatchOcc procedure to find occurrences of well-formed patterns

```
Function MatchOcc($\mathtt{Occ}_1, \dots, \mathtt{Occ}_k$):
    Input: $\mathtt{Occ}_1, \dots, \mathtt{Occ}_k$: set of occurrences of well-formed star free patterns $\alpha_1, \dots, \alpha_k$ over the same
           document. Each occurrence $o$ in $\mathtt{Occ}_i$ stores the position $o.pos$ of the document where the
           occurrence is located in the string $s$ and the starting and ending positions ($o.begin$ and $o.end$)
           of the occurrence in the document
    Output: $R_{\mathfrak{p}}$: set of occurrences of the well-formed pattern $p = \alpha_1 * \cdots * \alpha_k$ over the same document
1   begin
2       for $i \leftarrow 1$ to $k$ do
3           SortOccByEnd($\mathtt{Occ}_i$)
4       foreach $o_1 \in \mathtt{Occ}_1$ do
5           $\mathtt{head} \leftarrow o_1.end$
6           for $i \leftarrow 2$ to $k$ do
7               foreach $o \in \mathtt{Occ}_i$ do
8                   if $o.begin > \mathtt{head}$ then
9                       $\mathtt{head} \leftarrow o.end$, break
10                  DeleteOcc($o$)
11              if $\mathtt{Occ}_i = \emptyset$ then
12                  return $R_{\mathfrak{p}}$
13          $R_{\mathfrak{p}} = R_{\mathfrak{p}} \cup (o_1.pos, o_1.begin)$
14      return $R_{\mathfrak{p}}$
```

In Alg. 3.8, we report an improved version of MatchOcc that reduces the computational cost to $O(o_{\alpha,j}(\log(o_{\alpha,j})+k))$. This procedure relies on the fact that it is possible to build the occurrences of pattern $\mathfrak{p}$ in the document D_j much more efficiently if each of the k sets $\texttt{Occ}_1^j, \ldots, \texttt{Occ}_k^j$ is sorted in ascending order according to the ending positions of the occurrences in it. We now describe how the MATCHOCC procedure efficiently finds all the occurrences of $\mathfrak{p}$ over document D_j from the k sorted sets $\texttt{Occ}_1^j, \ldots \texttt{Occ}_k^j$; then, we prove its correctness. After sorting all the sets $\texttt{Occ}_1^j, \ldots \texttt{Occ}_k^j$ (line 3), the MATCHOCC procedure tries to build a set $\mathfrak{o}_{k,j}^h$ for any occurrence $o_{1,j}^h \in \texttt{Occ}_1^j$, $1 \leq h \leq |\texttt{Occ}_1^j|$ (lines 4-13). Specifically, for each set $\texttt{Occ}_i^j$, $i \in \{2, \ldots, k\}$, it finds (lines 7-10) the first occurrence satisfying $o_{i,j}.begin > o_{i-1,j}.end$ ($o_{i-1,j}.end$ is stored in variable $\texttt{head}$ in Alg. 3.8). Any occurrence $o_{i,j}$ such that $o_{i,j}.begin \leq \texttt{head}$ is erased from set $\texttt{Occ}_i^j$ (line 10); if no occurrence $o_{i,j}$ with $o_{i,j}.begin > \texttt{head}$ can be found (line 11), no more occurrences of $\mathfrak{p}$ in document D_j can be found (line 12). Conversely, in case an occurrence $o_{i,j}$ is found for every set $\texttt{Occ}_i^j$, $i \in \{2, \ldots, k\}$, then the set $\mathfrak{o}_{k,j}^h$ can be built and thus the occurrence identified by this set, which starts in position $o_{1,j}^h.begin$, is added to $R_{\mathfrak{p}}^j$ (line 13), that is the set of occurrences of the well-formed pattern $\mathfrak{p}$ over the document D_j.

We now prove that MATCHOCC allows to find all and only the occurrences of $\mathfrak{p}$ in a document D_j. The set $R_{\mathfrak{p}}^j$ computed by the MATCHOCC procedure contains only occurrences of $\mathfrak{p}$ over document D_j: indeed, an entry is added to this set if and only if a set $\mathfrak{o}_{k,j}^h = \{o_{1,j}, \ldots, o_{k,j}\}$, $1 \leq h \leq |\texttt{Occ}_1^j|$, of occurrences of $\alpha_1, \ldots, \alpha_k$, with $o_{1,j} = o_{1,j}^h$ and $o_{i,j}.end < o_{i+1,j}.begin$, for every $i < k$, is found. We now prove that the procedure finds all the occurrences. In particular, we want to show these two facts: an occurrence $o_{i,j}$ is erased from set $\texttt{Occ}_i^j$ (line 10) only if $o_{i,j}$ cannot belong to any other set $\mathfrak{o}_{k,j}^h$ besides the ones already found; if, for any of the sets $\texttt{Occ}_2^j, \ldots, \texttt{Occ}_k^j$, all occurrences in the set at hand are erased (line 12), then there are no more sets $\mathfrak{o}_{k,j}^h$ (and thus no more occurrences of $\mathfrak{p}$) to be found.

We start by proving the following property:

Lemma 3.2. *Consider the $k \geq 1$ sets of occurrences $\texttt{Occ}_1^j, \ldots, \texttt{Occ}_k^j$ of well-formed star-free patterns $\alpha_1, \ldots, \alpha_k$ over document D_j, with the set $\texttt{Occ}_i^j$, $i \in \{1, \ldots, k\}$, being sorted according to the ending position of each of its occurrences. If, for an occurrence $o_{i,j} \in \texttt{Occ}_i^j$, $i \in \{2, \ldots, k\}$, there is no set $\mathfrak{o}_{i,j}^h = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o_{1,j}^h$, such that $\forall i' < i, o_{i',j}.end < o_{i'+1,j}.begin$, then, for any $h' \geq h$, there is no set $\mathfrak{o}_{i,j}^{h'} = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o_{1,j}^{h'}$, such that $\forall i' < i, o_{i',j}.end < o_{i'+1,j}.begin$*

Proof. Assume that the thesis of the Lemma is false. This implies that, even if there is no set $\mathfrak{o}_{i,j}^h = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o_{1,j}^h$, for an occurrence $o_{i,j} \in \texttt{Occ}_i^j$, there exists a set $\mathfrak{o}_{i,j}^{h'} = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o_{1,j}^{h'}$, for an $h' > h$. Given the occurrence $o_{2,j}$ in $\mathfrak{o}_{i,j}^{h'}$, it holds that $o_{2,j}.begin > o_{1,j}^{h'}.end \geq o_{1,j}^h.end$, since the set $\texttt{Occ}_1^j$ is sorted according to the ending position of its occurrences; therefore, by replacing $o_{1,j}^{h'}$ with $o_{1,j}^h$ in $\mathfrak{o}_{i,j}^{h'}$, we obtain a set $\mathfrak{o}_{i,j}^h = \{o_{1,j}, \ldots, o_{i,j}\}$, with $o_{1,j} = o_{1,j}^h$, such that $\forall i' < i, o_{i',j}.end < o_{i'+1,j}.begin$. This contradicts the hypothesis that the set $\mathfrak{o}_{i,j}^h$ does not exist, so we conclude that there is no set $\mathfrak{o}_{i,j}^{h'}$ for the occurrence $o_{i,j} \in \texttt{Occ}_i^j$ for any $h' \geq h$. □

Lemma 3.3. *If an occurrence $o_{i,j} \in \texttt{Occ}_i^j$ is erased at line 10 of the* MATCHOCC *procedure reported in Alg. 3.8 in the h-th iteration of the loop at lines 4-13, then,* *for any $h' \geq h$, there is no set $\mathfrak{o}^{h'}_{i,j} = \{o_{1,j}, \dots, o_{i,j}\}$, with $o_{1,j} = o^{h'}_{1,j}$, such that $\forall i' < i, o_{i',j}.end < o_{i'+1,j}.begin$.*

Proof. We prove the lemma by induction over the sets $\texttt{Occ}^j_2, \dots, \texttt{Occ}^j_k$. We start with $\texttt{Occ}^j_2$. Suppose that an occurrence $o_{2,j} \in \texttt{Occ}^j_2$ is erased by MATCHOCC procedure in the h-iteration of the loop at lines 4-13: then, it means that $o_{2,j}.begin \leq \texttt{head}$, where $\texttt{head} = o^h_{1,j}.end$; thus, it immediately follows that $\mathfrak{o}^h_{2,j} = \{o^h_{1,j}, \dots, o_{2,j}\}$ cannot exist. Therefore, by Lemma 3.2, for any $h' \geq h$, no set $\mathfrak{o}^{h'}_{2,j} = \{o^{h'}_{1,j}, \dots, o_{2,j}\}$ exists. We now look at the general case for any occurrence $o_{i,j}$ in the set $\texttt{Occ}^j_i$ erased in the h-th iteration of loop at lines 4-13. Our inductive hypothesis is that for any element $o_{i-1,j} \in \texttt{Occ}^j_{i-1}$ already erased, there is no set $\mathfrak{o}^h_{i-1,j} = \{o^h_{1,j}, \dots, o_{i-1,j}\}$. If an occurrence $o_{i,j} \in \texttt{Occ}^j_i$ is erased, then it means that there is an occurrence $o'_{i-1,j} \in \texttt{Occ}^j_{i-1}$ (which is the first non erased occurrence in $\texttt{Occ}^j_{i-1}$) such that $o_{i,j}.begin \leq o'_{i-1,j}.end$; therefore, no set $\mathfrak{o}^h_{i,j} = \{o^h_{1,j}, \dots, o'_{i-1,j}, o_{i,j}\}$ exists. If we consider any $o_{i-1,j}$ already erased from $\texttt{Occ}^j_{i-1}$, then by inductive hypothesis there is no set $\mathfrak{o}^h_{i-1,j} = \{o^h_{1,j}, \dots, o_{i-1,j}\}$, which implies that there is also no set $\mathfrak{o}^h_{i,j} = \{o^h_{1,j}, \dots, o_{i-1,j}, o_{i,j}\}$. If we consider any $o_{i-1,j} \neq o'_{i-1,j}$ still in $\texttt{Occ}^j_{i-1}$, then $o_{i-1,j}.end \geq o'_{i-1,j}.end$, as the set $\texttt{Occ}^j_{i-1}$ is sorted in ascending order according to the ending position of its occurrences; therefore, $o_{i,j}.begin \leq o'_{i-1,j}.end \leq o_{i-1,j}.end$, which means that no set $\mathfrak{o}^h_{i,j} = \{o^h_{1,j}, \dots, o_{i-1,j}, o_{i,j}\}$ exists. In conclusion, there is no element in $o_{i-1,j} \in \texttt{Occ}^j_{i-1}$ such that it is possible to construct a set $\mathfrak{o}^h_{i,j} = \{o^h_{1,j}, \dots, o_{i-1,j}, o_{i,j}\}$, therefore, this set does not exist for the occurrence $o_{i,j}$. By applying Lemma 3.2, we can generalize this result to any $h' > h$. □

Lemma 3.3 implies that when an occurrence $o_{i,j}$ is erased in the h-th iteration of the loop at lines 4-13, then $o_{i,j}$ cannot belong to any set $\mathfrak{o}^{h'}_{k,j}$ for any $h' \geq h$. As an occurrence of $\mathfrak{p}$ over document D_j is identified by a set $\mathfrak{o}^h_{k,j} = \{o_{1,j}, \dots, o_{k,j}\}$, with $o_{1,j} = o^h_{1,j}$, of k occurrences from $\texttt{Occ}^j_1, \dots, \texttt{Occ}^j_k$, respectively, this means that occurrence $o_{i,j}$ cannot belong to any occurrence of $\mathfrak{p}$ identified by a set $\mathfrak{o}^{h'}_{k,j}$ for any $h' \geq h$. Since all the occurrences of $\mathfrak{p}$ identified by sets $\mathfrak{o}^{h'}_{k,j}$, for $h' < h$, have been already found at iteration h of the aforementioned loop, then the occurrence $o_{i,j}$ cannot belong to any other occurrence of $\mathfrak{p}$ besides the ones already found. Therefore, $o_{i,j}$ can be safely erased from its set.

Furthermore, if all occurrences are erased in the h-th iteration of the loop at lines 4-13 from a set $\texttt{Occ}^j_i$, $i \in \{2, \dots, k\}$, then all these occurrences by Lemma 3.3 cannot belong to any set $\mathfrak{o}^{h'}_{k,j} = \mathfrak{o}^{h'}_{k,j}$ for any $h' \geq h$. Since, by definition, a set $\mathfrak{o}^h_{k,j} = \{o_{1,j}, \dots, o_{k,j}\}$, with $o_{1,j} = o^h_{1,j}$, identifying an occurrence of $\mathfrak{p}$ over the document D_j is composed by k occurrences, one for each set $\texttt{Occ}^j_i$, then any set $\mathfrak{o}^{h'}_{k,j}$, for $h' \geq h$, cannot be built. Since all the sets $\mathfrak{o}^{h'}_{k,j}$, for $h' < h$, have been already built at the h-th iteration, there are no more occurrences of $\mathfrak{p}$ to be found, which means that the MATCHOCC procedure can immediately stop at line 12 returning the set $R^j_{\mathfrak{p}}$ of occurrences of $\mathfrak{p}$ in document D_j found so far.

Finally, we analyze the computational and communication costs of the QUERYPATTERN procedure in Alg. 3.7, for a pattern $\mathfrak{p}$ with $k \geq 1$ well-formed star-free patterns $\alpha_1, \dots, \alpha_k$. To this extent, we denote as m_{α_i} (resp. o_{α_i}), $i \in \{1, \dots, k\}$, the length (resp. number of occurrences over the document collection $\mathbf{D}$) of the well-formed star-free pattern α_i , and by m_α (resp. o_α) the sum of all of these lengths (resp. occurrences), that is $m_\alpha = \sum_{i=1}^{k} m_{\alpha_i}$ (resp. $o_\alpha = \sum_{i=1}^{k} o_{\alpha_i}$)). In case these patterns contain wildcard characters, the computational cost at server side amounts to $\mathrm{O}((m_\alpha{+}o_\alpha)\frac{n}{b}\log^3(N))$, due to the k executions of the STARFREEQUERY procedure, each one with cost $\mathrm{O}((m_{\alpha_i} + o_{\alpha_i})\frac{n}{b}\log^3(N))$; similarly, the communication cost amounts to $\mathrm{O}(b\log^2(n)\log(N)\sum_{i=1}^{k} m_{\alpha_i}{\cdot}o_{\alpha_i})$, as each of the k STARFREEQUERY procedures exhibits $\mathrm{O}(b\log^2(n)\log(N)m_{\alpha_i}o_{\alpha_i})$ communication cost. Remarkably, in case all the patterns $\alpha_1,\dots,\alpha_k$ do not include wildcard characters, then the computational cost at server side remains independent from the number of occurrences, while the communication cost is reduced to $\mathrm{O}((m_\alpha{+}o_\alpha)b\log^2(n)\log(N))$, because each of the k STARFREEQUERY procedures simply executes the QUERY algorithm.

3.3 Security Analysis of Our Multi-User PPSS Protocol

We now show that our PPSS protocol ensures the confidentiality of the remotely stored string, of the searched substring, and of the results returned by each search query, as well as guaranteeing the privacy of both the search and access patterns. In the following, adopting the framework introduced by Curtmola in [43], we provide a formal definition of the information leakage coming from a PPSS protocol and we formally specify the adversarial model as well as the security guarantees provided by our PPSS solution. We first consider the basic version of our PPSS protocol, and then we discuss the security guarantees of the multi-user extension and of our privacy-preserving queries for string containing wildcard characters. In our PPSS protocol, some parameters, such as the sample period P employed to construct the ABWT and the radix b employed for Lipmaa's PIR protocol, are chosen only for performance reasons and independently from sensitive data; therefore, as the knowledge of these parameters does not affect the security guarantees of our PPSS protocol, we assume that the same value for these parameters is always employed, in turn simplifying our security proof.

We start by defining the leakage of a PPSS protocol. Informally, the leakage amounts to all the information about the confidential data involved in the protocol that the adversary can infer by observing the execution of the PPSS protocol. In particular, all the information observed by the adversary during the execution of the protocol is referred to as its *trace*. In our threat model, we assume that the adversary is not able to compromise the data owner or the authorized users performing queries, as otherwise the security guarantees of the protocol are trivially subverted. Conversely, we assume that the server is controlled by a semi-honest adversary, who does not misbehave in the protocol but it is willing to learn as much information as possible. Given these assumptions, the trace of a PPSS protocol is given only by the data exchanged between the client and the remote server and from the computation run at server side. We formally state the trace and the leakage of a PPSS protocol in the following definition.

Definition 3.5 (Trace and Leakage of PPSS Protocol)**.** *Given a document collection* $\mathbf{D}$*,* $d{\geq}1$ *strings* $q_1,\dots,q_d$*, and a PPSS protocol* $\mathcal{P} = (\text{SETUP}, \text{QUERY})$*, its trace* $\mathcal{T} = (\mathcal{T}_{\mathbf{D}}, \mathcal{T}_{q_1},\dots,\mathcal{T}_{q_d})$ *is defined as follows.*

- $\mathcal{T}_{\mathbf{D}} = \langle \mathbf{D} \rangle$ *represents the information observed by the adversary in the* SETUP

```
**Experiment** $transcript \leftarrow \mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)$:
$(\mathbf{D}, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{\mathbf{D}}(1^{\lambda})$
$(\langle\mathbf{D}\rangle, aux_s) \leftarrow \mathcal{P}.\mathtt{Setup}(\mathbf{D}, 1^{\lambda})$
$\forall i \in \{1, \ldots, d\}$:
    $(q_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i\left(\{\mathcal{T}_{\mathbf{D}}, \mathcal{T}_{q_1}, \ldots, \mathcal{T}_{q_{i-1}}\}, \mathtt{st}_{\mathcal{A}}\right)$
    $\forall j \in \{1, \ldots, w\}$:
        $\langle q_i\rangle_j \leftarrow \mathcal{P}.\mathtt{Trapdoor}(j, q_i, aux_s, res_1, \ldots, res_{j-1})$
        $(\langle res_j\rangle, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}.\mathtt{Search}(\mathtt{st}_{\mathcal{A}}, \langle q_i\rangle_j, \langle\mathbf{D}\rangle)$
        $res_j \leftarrow \mathcal{P}.\mathtt{Retrieve}(\langle res_j\rangle, aux_s)$
$transcript \leftarrow \{\mathtt{st}_{\mathcal{A}}, \mathcal{T}_D, \mathcal{T}_{q_1}, \ldots, \mathcal{T}_{q_d}\}$

**Experiment** $transcript \leftarrow \mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$:
$(\mathbf{D}, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{\mathbf{D}}(1^{\lambda})$
$(\langle\mathbf{D}\rangle, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_{\mathbf{D}}(\mathcal{L}_{\mathbf{D}}, 1^{\lambda})$
$\forall i \in \{1, \ldots, d\}$:
    $(q_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i\left(\{\mathcal{T}_{\mathbf{D}}, \mathcal{T}_{q_1}, \ldots, \mathcal{T}_{q_{i-1}}\}, \mathtt{st}_{\mathcal{A}}\right)$
    $\forall j \in \{1, \ldots, w\}$:
        $(\langle q_i\rangle_j, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_{q_i}(j, \mathtt{st}_{\mathcal{S}}, \{\mathcal{L}_{\mathbf{D}}, \mathcal{L}_{q_1}, \ldots, \mathcal{L}_{q_i}\})$
        $(\langle res_j\rangle, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}.\mathtt{Search}(\mathtt{st}_{\mathcal{A}}, \langle q_i\rangle_j, \langle\mathbf{D}\rangle)$
$transcript \leftarrow \{\mathtt{st}_{\mathcal{A}}, \mathcal{T}_D, \mathcal{T}_{q_1}, \ldots, \mathcal{T}_{q_d}\}$
```

Figure 3.1: *Security game experiments for our PPSS protocol*

phase, which is limited to the outsourced privacy-preserving representation $\langle\mathbf{D}\rangle$ of the document collection.

- *$\mathcal{T}_{q_i}, i \in \{1, \ldots, d\}$, represents the information observed by the adversary in the w iterations (rounds) executed during the* QUERY *phase of the protocol for the string q_i. Specifically, $\mathcal{T}_{q_i} = \{\langle q_i\rangle_1, \mathtt{st}_1, \langle res_1\rangle, \ldots, \langle q_i\rangle_w, \mathtt{st}_w, \langle res_w\rangle\}$, that is the trapdoors $\langle q_i\rangle_j$, $j \in \{1, \ldots, w\}$, computed in each round by the* TRAPDOOR *procedure, the states $\mathtt{st}_j$ that store all the information observed during the execution of the* SEARCH *procedure in each round, and the results $\langle res\rangle_j$ computed in each round by the* SEARCH *procedure.*

The leakage $\mathcal{L} = (\mathcal{L}_{\mathbf{D}}, \mathcal{L}_{q_1}, \ldots, \mathcal{L}_{q_d})$ of the protocol $\mathcal{P}$ is the information about the document collection $\mathbf{D}$, the d strings q_i and the results of the d queries for such strings inferred by the adversary from the trace $\mathcal{T}$.

The security game stated in Def. 3.6 allows to prove that a semi-honest adversary does not learn anything but the leakage $\mathcal{L}$. To this end, this definition requires the existence of a simulator $\mathcal{S}$, taking as inputs only $\mathcal{L}_{\mathbf{D}}$ and $\mathcal{L}_q = \{\mathcal{L}_{q_1}, \ldots, \mathcal{L}_{q_d}\}$, which is able to generate a *transcript* of the PPSS protocol for the adversary that is computationally indistinguishable from the one generated when a legitimate client interacts with the server during a real execution of the protocol. The transcript of the protocol is given by the trace observed by the adversary and any other information inferred by the adversary during the execution of the protocol, which is stored in a state $\mathtt{st}_{\mathcal{A}}$.

Definition 3.6 (Security Game)**.** *Given a PPSS protocol $\mathcal{P}$ with security parameter λ, $d \geq 1$ queries, the leakage $\mathcal{L}$ and the trace $\mathcal{T}$ of $\mathcal{P}$ as defined in Def. 3.5, an adversary $\mathcal{A}$ consisting of $d+1$ probabilistic polynomial time algorithms $\mathcal{A} = (\mathcal{A}_{\mathbf{D}}, \mathcal{A}_1, \ldots \mathcal{A}_d)$, and a simulator $\mathcal{S}$, which is also a tuple of $d+1$ probabilistic polynomial time algorithms $\mathcal{S} = (\mathcal{S}_{\mathbf{D}}, \mathcal{S}_{q_1}, \ldots \mathcal{S}_{q_d})$, the two probabilistic experiments $\mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$*

shown in Fig. 3.1 are considered. Denote as $\mathcal{D}(o)$ a probabilistic polynomial time algorithm taking as input a transcript of an experiment o and returning a boolean value indicating if the transcript belongs to the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)$ ($\mathcal{D}(o) = 0$) or $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ ($\mathcal{D}(o) = 1$) experiment. The protocol $\mathcal{P}$, with leakage $\mathcal{L}$, is secure against any semi-honest probabilistic polynomial time adversary $\mathcal{A} = (\mathcal{A}_{\mathbf{D}}, \ldots \mathcal{A}_d)$, if, for every such $\mathcal{A}$, there exists a simulator $\mathcal{S} = (\mathcal{S}_{\mathbf{D}}, \mathcal{S}_{q_1}, \ldots \mathcal{S}_{q_d})$ such that for every $\mathcal{D}$:

$$\mathtt{Pr}\Big(\mathcal{D}(o) = 1 \mid o \leftarrow \mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)\Big) - \mathtt{Pr}\Big(\mathcal{D}(o) = 1 \mid o \leftarrow \mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)\Big) \leq \epsilon(\lambda)$$

where $\epsilon(\cdot)$ is a negligible function.

In the experiments shown in Fig. 3.1, $\mathbf{D}$ is chosen by the adversarial algorithm $\mathcal{A}_{\mathbf{D}}$ and each query q_i, $1 \leq i \leq d$, is *adaptively* chosen by the i-th adversarial algorithm $\mathcal{A}_i$, depending on the transcript of the protocol in the previous queries.

The $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment represents an actual execution of the protocol, where the client receives the document collection $\mathbf{D}$ and the d queries and it behaves as specified in the protocol; conversely, in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment, the client is simulated by $\mathcal{S}$, which however employs only the leakage information $\mathcal{L} = (\mathcal{L}_{\mathbf{D}}, \mathcal{L}_{q_1}, \ldots, \mathcal{L}_{q_d})$. In particular, the simulator $\mathcal{S}_{\mathbf{D}}$ constructs a privacy-preserving representation $\langle \mathbf{D} \rangle$ by exploiting only the knowledge of $\mathcal{L}_{\mathbf{D}}$, while each simulator $\mathcal{S}_{q_i}$ constructs the trapdoor for each round of the i-th query by exploiting only the knowledge of the leakage $\mathcal{L}_{\mathbf{D}}$, $\mathcal{L}_{q_j}, 1 \leq j \leq i$. Therefore, the transcript of the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment depends only on the leakage $\mathcal{L}$, as the real document collection $\mathbf{D}$ and the strings $q_1, \ldots, q_d$ are never employed in the simulation of the protocol. Thus, if the transcript is computationally indistinguishable from the one of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, then it means that also this transcript reveals no more information than $\mathcal{L}$, as otherwise the additional information leakage of the real protocol would allow to distinguish between the transcripts of the two experiments. Since the transcript of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment corresponds to the information observed and derived by the adversary in our PPSS protocol, then no information other than $\mathcal{L}$ can be inferred from the adversary in our PPSS protocol, in turn proving that the protocol leaks no more information than $\mathcal{L}$ to the adversary.

In the following Thm. 3.1, we state the leakage $\mathcal{L}$ that makes our protocol secure according to Def. 3.6, which, as we have just discussed, corresponds to the leakage of our PPSS protocol.

Theorem 3.1. *Given a document collection $\mathbf{D}$ with $z \geq 1$ documents $\{D_1, \ldots, D_z\}$ and $d \geq 1$ substrings $q_1, \ldots, q_d$, our PPSS protocol is secure against a semi-honest adversary, as per Def. 3.6, with a leakage $\mathcal{L} = (\mathcal{L}_{\mathbf{D}}, \mathcal{L}_{q_1}, \ldots, \mathcal{L}_{q_d})$, where:*

- $\mathcal{L}_{\mathbf{D}} = \sum_{i=1}^{z}(\text{LEN}(D_i) + 1) = n$
- $\mathcal{L}_{q_i} = (\text{LEN}(q_i), |O_{\mathbf{D},q_i}|)$, $1 \leq i \leq d$, *where $O_{\mathbf{D},q_i}$ is defined as per Def. 1.5.*

The proof of this claim can be found in Appendix 2. Informally, the security of our protocol is based on the security of the PIR protocol being employed and on the semantic security of the encryption scheme used to compute the encrypted data structures $\langle A_P \rangle$ and $\langle Suf \rangle$, as the server observes only PIR queries on arrays encrypted with a semantically secure encryption scheme. The only information leaked to the server are the sizes of these data structures, which are both proportional to the size n of the document collection, the number of documents z in $\mathbf{D}$, which is trivially leaked by the encrypted documents outsourced to the server, the length m of the searched substring, leaked by the number of PIR queries performed over $\langle A_P \rangle$, and the number of occurrences o_q, which is proportional to the number of entries retrieved by the aggregate PIR queries over $\langle Suf \rangle$. We remark that Thm. 3.1 guarantees search and access pattern privacy, as they are not enclosed in the leakage $\mathcal{L}$.

Security Guarantees in Multi-User Scenario

We now informally analyze the security guarantees achieved by our PPSS protocol in a multi-user scenario. In such setting, we need to consider also the information that is leaked to the multiple users involved in the protocol. First of all, we observe that each user can easily reconstruct the entire document collection $\mathbf{D}$. Indeed, each user can perform $|\Sigma|$ queries through the PPSS protocol, each retrieving the occurrences of a distinct character in the alphabet Σ. From each of these queries, the user learns the positions of all the occurrences of a specific character in Σ and the document where each occurrence is located; by combining the information of all these queries, the users learn the character found in each position of each document in the collection, hereby entirely reconstructing $\mathbf{D}$. We observe that the users reconstruct the content of the documents relying only on the results of legitimate queries, which is an information available in any PPSS protocol providing the substring search functionality specified in Def. 1.5. In particular, we observe that such information leakage would occur even in the PPSS protocol with the highest possible privacy guarantees, where the queries are executed by a trusted party that reveals to the users only the results of their queries. Therefore, this leakage is not a security weakness of our protocol, but an unavoidable information leakage that the data owner must necessarily accept.

Furthermore, our solution guarantees that, given a query for a string q performed by a user, all the other users, even by observing the execution of the query at server side, can learn no more information than the leakage $\mathcal{L}_q$ reported in Thm. 3.1. Equivalently, a query issued by any user in our PPSS protocol exhibits the same information leakage to both the untrusted server and other authorized users in the protocol. This property stems from the security guarantees of the PIR protocol: indeed, as for the untrusted server, any other authorized user observes only a set of PIR queries from the victim user, thus the privacy guarantees of PIR ensures that no information about the entries retrieved by the victim user from the outsourced full-text index can be learned. In particular, with the Lipmaa's PIR employed in our PPSS protocol, each user employs its own key-pair for the DJ LFAHE scheme, which implies that all the parties observe only ciphertexts of a semantically secure encryption scheme that can be decrypted only by the user issuing the query. This property is crucial to discourage collusion between the untrusted server and authorized users in our PPSS protocol: indeed, although the server is interested in such a collusion as it allows to learn the content of the document collection, the authorized users have no incentive to collude with the server, as they cannot learn any meaningful information about queries of the other users.

Privacy Guarantees for Queries with Wildcards

To analyze the information leakage of privacy-preserving queries containing wildcard characters, we distinguish two possible use case scenarios: in the first one, which is more unlikely, the adversary knows that the client is performing a single query (e.g., an application scenario where each user is allowed to perform a single query per day); in the second one, the client may perform an arbitrary number of queries.

In the first scenario, the adversary can infer the number k of $*$ wildcards in the queried pattern $\mathfrak{p} = \alpha_1 * \cdots * \alpha_{k+1}$ and, for each of the $k{+}1$ well-formed star-free patterns $\alpha_1, \ldots, \alpha_{k+1}$, if there is at least a wildcard different from $\&$. The value k can be inferred by the number of private accesses to the encrypted SA $\langle Suf \rangle$ performed during the execution of the query: indeed the QUERYPATTERN procedure runs $k{+}1$ times (line 4 in Alg. 3.7) the STARFREEQUERY function reported in Alg. 3.6, which in turn accesses $\langle Suf \rangle$ only twice per run during the execution of the BATCHEDRETRIEVAL function (invoked either at line 13 in Alg. 3.6 or during the QUERY procedure invoked at line 4 in Alg. 3.6). Furthermore, if the client performs a private access to the encrypted array $\langle s \rangle$ (line 19 in Alg. 3.6) after the i-th access to $\langle Suf \rangle$, $1 \leq i \leq k{+}1$, then the adversary learns that the i-th well-formed star-free pattern α_i found in $\mathfrak{p}$ contains a wildcard character different from $\&$. Once the adversary has reconstructed the structure of $\mathfrak{p}$, for each well-formed star-free pattern with no wildcard character (except for $\&$) it learns its length and its number of occurrences, as they are leaked by the QUERY procedure invoked at line 4 in STARFREEQUERY (Alg. 3.6); otherwise, for each well-formed star-free pattern α_i with at least a wildcard character different from $\&$, it learns the length of the longest string in L_{α_i} (from the number of elements retrieved at line 19 in Alg. 3.6) and the upper bound o_{min} (see communication cost of STARFREEQUERY procedure in Sec. 3.2) on the number of its occurrences (from the number of elements retrieved at line 13 in Alg. 3.6). Therefore, the adversary cannot learn the exact number of occurrences of a pattern unless it is a well-formed star-free pattern with no wildcards other than $\&$.

In the second scenario, where the client may perform an arbitrary number of queries, the adversary can no longer reconstruct the number of $*$ wildcards: indeed, the adversary only observes a set of queries for well-formed star-free patterns, but it cannot determine which of them are portions of the same well-formed pattern. Nonetheless, the adversary can still infer, for each of the observed queries, if the queried well-formed star-free pattern $\mathfrak{p}$ contains at least a wildcard character other than $\&$ by verifying if the client retrieves any element from the encrypted array $\langle s \rangle$. It is worth noting that this information leakage may be not accurate for the adversary: indeed, although not specified in the functionality provided by a PPSS protocol, in some application scenarios the client, once determined the positions of an occurrence in a document, may need to download the portions of the document corresponding to such an occurrence, hereby privately accessing via PIR queries the encrypted array $\langle s \rangle$ too. Similarly to the previous scenario, in case the adversary determines that the searched well-formed star-free pattern $\mathfrak{p}$ contains at least a wildcard character other than $\&$, it learns the length of the longest string in $L_{\mathfrak{p}}$ and the upper bound o_{min} on the number of its occurrences; otherwise, it learns the length of $\mathfrak{p}$ and the number of its occurrences. Nonetheless, since in this scenario the client cannot know if $\mathfrak{p}$ is a portion of a bigger well-formed pattern or not, the adversary can never know with certainty both the length and the number of occurrences of a well-formed pattern.

The leakage of the structure of the well-formed pattern in both scenarios can be prevented with the following two modifications: the first one consists of conceiving the encrypted arrays $\langle A_P \rangle$, $\langle Suf \rangle$ and $\langle s \rangle$ in $\langle \mathbf{D} \rangle$ as a single dataset, thus performing any PIR query over all these three arrays; the second modification requires that PIR queries always retrieve elements in batches of constant size (otherwise the adversary can infer which array among $\langle A_P \rangle$, $\langle Suf \rangle$ and $\langle s \rangle$ is accessed from the number of retrieved elements). Nonetheless, these two modifications would introduce a significant performance overhead to the PPSS protocol, as the performance benefits of implementing a batching retrieval would be lost and also the performance penalty due to accessing with a PIR protocol a much larger dataset must be kept into account. Therefore, we deem the leakage related to the execution of queries with wildcards acceptable in most of the practical scenarios and recommend the described countermeasure only for specific use cases where the information leakage about the structure of the queried well-formed pattern is actually a sensitive data.

3.4 Experimental Evaluation of our PPSS Protocol

In this section, we report the experiments that we performed in order to evaluate the actual performance and the practicality of our multi-user PPSS protocol on a real world genomics use case. We run our tests on a dual Intel Xeon CPU E5-2620 clocked at 3 GHz, endowed with 128 GiB DDR4-2133, and 64-bit Gentoo Linux17.0 OS. For our experimental campaign, we built a C/C++ implementation of our PPSS protocol, providing a cryptographic security level of at least $\lambda = 80$ bits. To implement the PIR-related cryptographic operations, our implementation relies on both the multi-precision integer arithmetic GMP library [66] and a proper parametrization of the DJ scheme implementation provided by the LIBHCS library [150]. For the cell-wise encryption/decryption of $\langle \mathbf{D} \rangle = (\langle A_P \rangle, \langle Suf \rangle, \langle s \rangle)$, our implementation employs the AES-128 symmetric encryption algorithm with CTR mode available in the OPENSSL library [117] (ver. 1.0.2r).

We chose as our case study a genomics dataset in the widely employed FASTA format [38]; although this format may employ up to 20 characters to represent DNA sequences, only 5 symbols are employed in our dataset, i.e.: $\Sigma = \{C, G, A, T, N\}$. Our genomics dataset is a single document containing approximately $40 \cdot 10^6$ nucleotides (characters) belonging to the 21^{st} human chromosome selected from the ENSEMBL publicly available data [59]. In the experiments, we considered documents with variable sizes replicating and truncating the mentioned dataset appropriately. We considered substring searches with a substring q having $m = 6$ characters, as it is the size of many *restriction enzyme sites* (transcribed as m-character strings) that are commonly employed in DNA-based paternity tests. Indeed, the test employs the distances between the occurrences of one of the mentioned substrings in the DNA fragments of two hosts to identify if the hosts are related [7].

For the construction of the ABWT A_P in the implementation of our PPSS protocol, we chose a sample period P that made the size of each entry approximately $\log(N)$ bits wide, with N being the modulus chosen in the DJ LFAHE scheme employed in Lipmaa's PIR. In this way, we reduce the number of entries of the encrypted array $\langle A_P \rangle$ as much as possible without increasing the bandwidth, as the entry still fits in a single ciphertext of the DJ LFAHE scheme. Lowering the number of entries is beneficial for the performance of the Lipmaa's PIR queries, as it allows to reduce the number

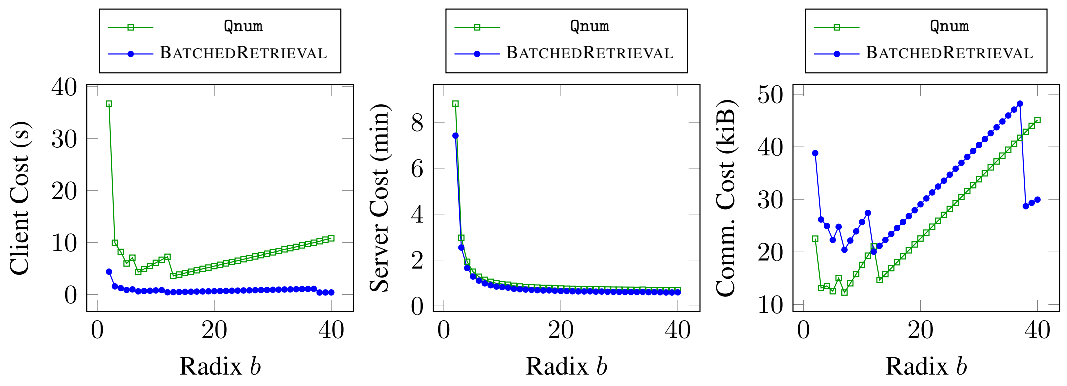


Figure 3.2: *Performance of our PPSS protocol as a function of the radix b employed in the PIR algorithms. We consider the retrieval of the positions of the 248 occurrences of $q = CTGCAG$ in a genome with $500k$ nucleotides*

of homomorphic operations in the PIR-Search procedure. We employed a similar optimization for the construction of the encrypted SA $\langle Suf \rangle$, as we encrypted in a single AES ciphertext of approximately $\log(N)$ bits as many entries as possible from the SA Suf.

We start the discussion of the results of our experimental campaign with the performance evaluation of the Query procedure. We evaluated separately the first phase of the Query procedure, labeled as `Qnum` in Alg. 3.2, which computes the number of occurrences of the searched substring, and the BatchedRetrieval procedure reported in Alg. 3.4, which retrieves the set of positions of the leading character of the occurrences. We remark that the communication cost reported in our results refer to a single round of communication. In Fig. 3.2, we show how the performance of the two phases of the Query procedure are affected by the radix b employed in the Lipmaa's PIR algorithm. As expected, increasing values of b allows to significantly decrease the computational cost on server side; conversely, the client and communication costs, which include a factor $O(b \cdot poly(\log_b(n)))$ (see Sec. 1.5), increase with the values of b, save for small values of b. The results suggest that the optimal value of b must be found considering the overall response time of a query, and should be differentiated between the `Qnum` phase and the BatchedRetrieval procedure as b_n and b_r, respectively.

In the next batch of tests, we considered a single-core implementation where we employed the same value $b = 20$ for genomes of increasing size, to observe how the performances are affected only by the size of the document collection. In addition, we considered also a multi-core implementation of the PIR-Search procedure of the Lipmaa's PIR protocol. Specifically, we employed a simple parallelization strategy that hinges upon b cores to simultaneously compute all the b recursive calls of Alg. 3.5. For these tests, we employed the optimal values b_n and b_r for each document size. In order to focus our analysis on the dependence between the performance metrics of our implementations and the size of the document collection, in these tests the BatchedRetrieval procedure retrieved a single occurrence: indeed, since the number of occurrences may increase proportionally to the size of the document collection, the performance metrics would be affected also by the progressively higher number of occurrences to be retrieved; we will separately evaluate in subsequent experiments the dependence of these metrics on the number of occurrences retrieved by the client. The

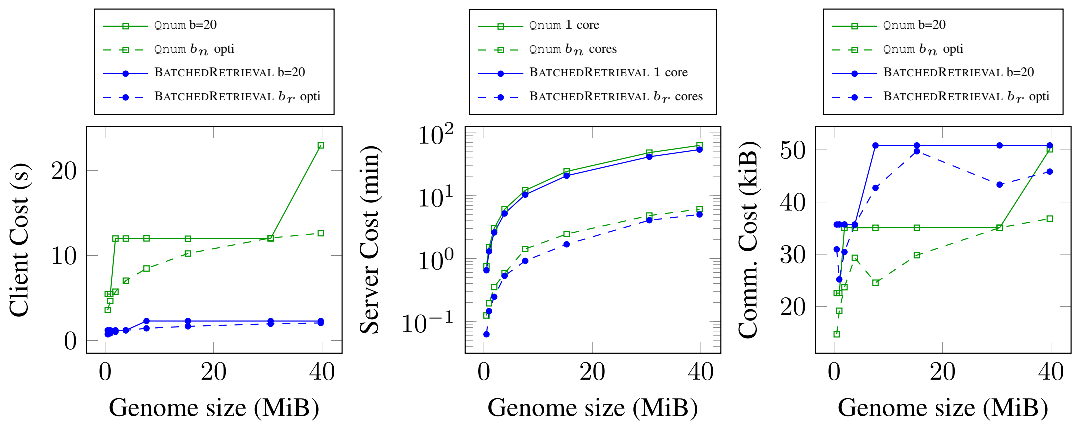


Figure 3.3: *Performance of our PPSS protocol as a function of the genome document size to find one occurrence of the substring $q = CTGCAG$. Considering each document size in increasing order, the optimal values of radixes b_n and b_r employed during the experiments are* $\{13, 17, 21, 26, 14, 17, 20, 21\}$ *and* $\{27, 14, 17, 20, 24, 28, 17, 18\}$*, respectively.*

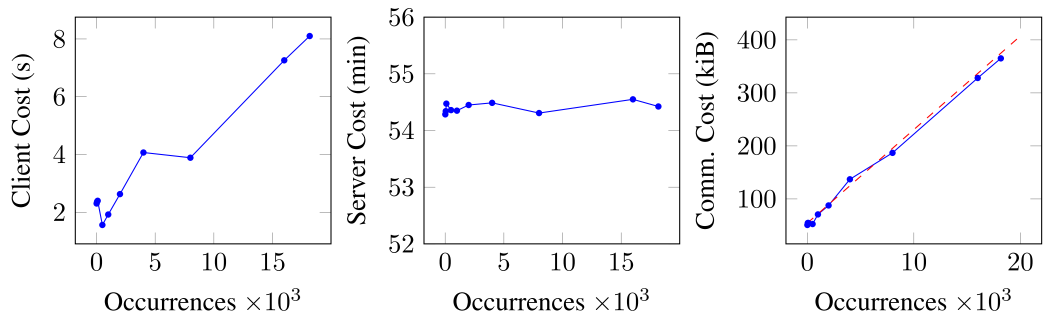


Figure 3.4: *Performance of* BatchedRetrieval *procedure in our PPSS protocol as a function of the number of occurrences retrieved by the client. We consider the retrieval of a subset of the* 18168 *occurrences of $q = CTGCAG$ over the entire 21st human chromosome with a single core implementation. The dashed red line represents the linear model* $Comm.Cost = 17.65 * Occurrences + 54218$ *that fits the communication cost growth.*

results of these tests are shown in Fig. 3.3. Regarding the server cost, we observe a linear trend in both the single-core (continuous lines in Fig. 3.3) and the multi-core implementations (dashed lines in Fig. 3.3); nevertheless, the multi-core implementation is at least one order of magnitude faster than the single-core, achieving much more practical performances (i.e., approximately 5 minutes to search for the substring $q = CTGCAG$ in a $40 \cdot 10^6$ characters document containing the whole chromosome).

The client and communication costs show the expected poly-logarithmic trend which allows to exchange only few tens of kilobytes of data to search for the occurrences of $q = CTGCAG$ in the whole chromosome. Furthermore, in Fig. 3.3 the dashed lines on plots reporting the client and communication costs show the benefits of employing specific values b_n and b_r tailored for the size of the document.

In Fig. 3.4, we report the evaluation of the performance metrics of our PPSS protocol for an increasing number of occurrences retrieved by the user issuing the query. We observe that both the computational cost at client side and the communication one increase linearly with the number of occurrences, which is expected as the client re-

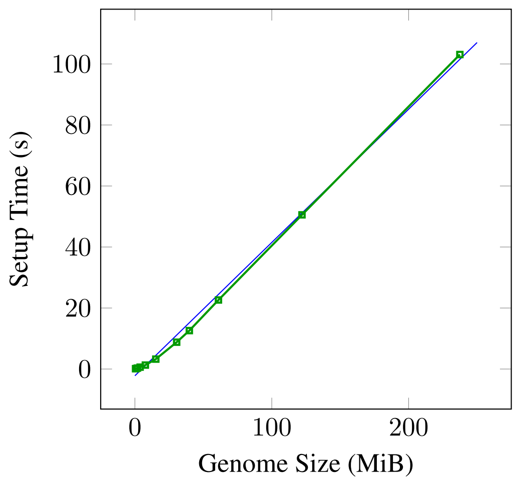


Figure 3.5: *Execution time of the* SETUP *procedure for genomes of increasing size. The blue line shows the fit between the linear model given by* $SetupTime = 0.4369 * GenomeSize - 2.2$ *and the experimental data*

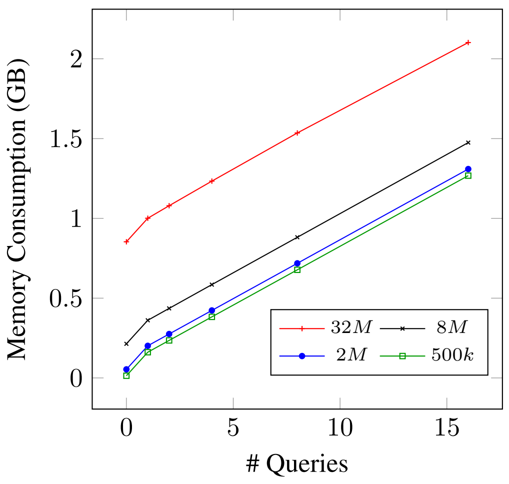


Figure 3.6: *Memory consumption of our PPSS protocol when multiple simultaneous queries are performed. Each line represent a genome with a different size*

ceives and decrypts each of these occurrences. Remarkably, the linear regression model that fits the communication cost shows that each additional occurrence retrieved by the client requires only 18 B of bandwidth. The results of our evaluation confirm that the computational cost at server side is basically independent from the number of retrieved occurrences. Indeed, the BATCHEDRETRIEVAL procedure shows approximately the same execution time both to retrieve few occurrences and to retrieve all the 18168 occurrences of the string $q = CTGCAG$ over the entire 21^{st} chromosome.

Willing to compare the execution time of our PPSS protocol with the one of a baseline solution with no security guarantees, we considered the backward-search procedure outlined in Alg. 1.2, employing the ABWT-based algorithm to compute the RANK procedure. Specifically, we measured the response time of a query for a single occurrence of the substring $q = CTGCAG$ in the outsourced document. The result of this experiment showed a response time for the backward-search equal to a few microseconds. We remark that a significant portion of this overhead can be motivated by the computational complexity of the QUERY procedure in our PPSS protocol, which depends linearly on the size of the outsourced document; conversely, the computational complexity of the backward-search method is unrelated to the size of the outsourced document in case of querying for a single occurrence of q.

After extensively analyzing the QUERY procedure, we report in Fig. 3.5 the execution time for genomes of increasing size of the SETUP procedure in Alg. 3.1, which builds the privacy-preserving representation $\langle \mathbf{D} \rangle$ of the dataset. In this test we considered also the genomics data corresponding to the 1^{st} human chromosome, which is much bigger than the 21^{st} one employed in all other tests. The experimental results confirm the expected linear trend and they show practical performance for the SETUP procedure: indeed, building the privacy-preserving representation of the 1^{st} human chromosome, which is as big as 238 MB, requires only 103 seconds.

Lastly, willing to verify the limited memory consumption when multiple-queries are simultaneously performed, we run each query in a separate thread, measuring the memory consumption of the process, as exposed by the process record in Linux's `proc` virtual filesystem. Figure 3.6 shows that as the number of simultaneous queries is increased, the memory consumption increases keeping (roughly) the same rate for the four dataset sizes considered. These results confirm the sublinear amount of additional memory required by each simultaneous query and the substantial storage savings given by our optimized PIR-SEARCH procedure over the naive strategy that replicates the whole dataset per-query.

Evaluation of Queries with Wildcards

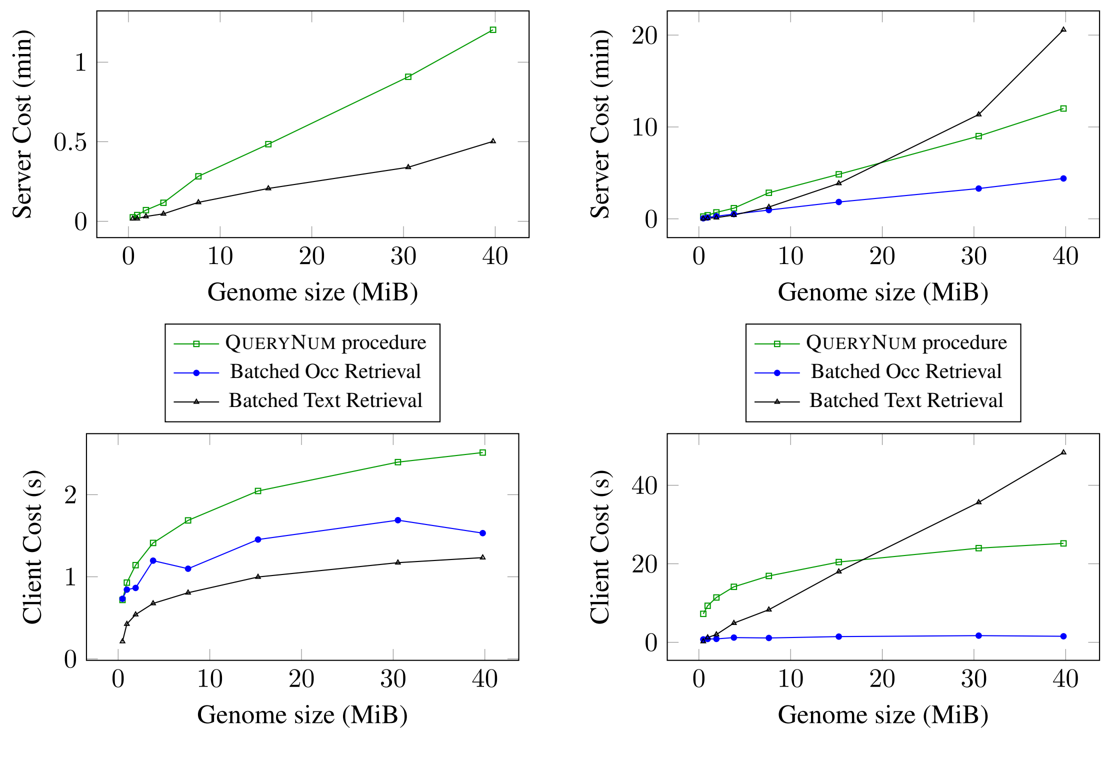
We also implemented the STARFREEQUERY and the QUERYPATTERN procedures, with the aim of experimentally validating their correctness as well as evaluating their performance. In our implementation of the STARFREEQUERY procedure, we relied on the Perl Compatible Regular Expressions (PCRE) library [71] (ver. 10.35) for the MATCHSHORTESTPREFIX procedure (line 21 in Alg. 3.6), which matches the portions of the string s privately retrieved from the outsourced encrypted array $\langle s \rangle$ (line 19). In particular, we employ the option `PCRE_UNGREEDY` to find the shortest match of a pattern instead of the longest one, which is the default behavior of PCRE library. Similarly to the construction of the outsourced suffix array, we packed in a single ciphertext of $O(\log(N))$ bits as many characters as possible from s, hence reducing as much as possible the number of entries of the array $\langle s \rangle$. In our evaluation, we employed a parallel implementation of the PIR-SEARCH procedures on server side.

In the evaluation of STARFREEQUERY procedure, we considered the following well-formed star-free pattern, which is decomposed according to Lemma 3.1 in three wildcard-free substrings (highlighted in red) and 5 substrings containing wildcards (highlighted in blue):

$$\mathfrak{p} = ?(GC|A|)GCCTATCG(G|TAC|??)([!CT]?|)TA?(TG|CGT|TA|[ACG][ATG])GTC(|?)$$

Such pattern allows to fully validate the capability of our privacy-preserving STARFREEQUERY procedure, as it includes all the legit wildcards defined in our format while reasonably representing the type of well-formed star-free patterns that may be matched in our PPSS protocol. Indeed, this pattern matches substrings with length ranging from 18 to 26 characters, with the longest matches including 50% of wildcard characters. The issuing of queries with patterns that matches more than 50% of the characters through wildcards is not representative of the usual application scenarios.

The computational costs of the STARFREEQUERY procedure for different dataset sizes are reported in Fig. 3.7. In our evaluation, we split the computational and communication costs according to the three most computationally intensive operations of the STARFREEQUERY procedure: the computation of the number of occurrences of each of the k wildcard-free substrings of the searched well-formed star-free pattern $\mathfrak{p}$, performed with the k executions of the QUERYNUM procedure (line 10 in Alg. 3.6); the batched retrieval of the o_{min} occurrences of the wildcard-free substring with the least number of occurrences (line 13), labeled as *Batched Occ Retrieval* in Fig. 3.7; the batched retrieval of the portions of the string s where the occurrences of $\mathfrak{p}$ can



(a) *Unitary costs* **(b)** *Overall costs*

Figure 3.7: *Computational costs of a multi-core implementation of* StarFreeQuery *procedure of our PPSS protocol as a function of the genome size. The tested query retrieves all the occurrences of the well-formed star-free pattern* $\mathfrak{p} = ?(GC|A|)GCCTATCG(G|TAC|??)([!CT]?|)TA$ $?(TG|CGT|TA|[ACG][ATG])GTC(|?)$

be found (line 19), labeled as *Batched Text Retrieval* in Fig. 3.7. Since the costs of the QueryNum procedure depend on the number of characters of the k wildcard-free substrings of the pattern $\mathfrak{p}$, and the costs of the *Batched Text Retrieval* depend on the number o_{min} of portions of s that must be retrieved, we report in Fig. 3.7(a) also the unitary costs for both these operations; conversely, since the server cost of the *Batched Occ Retrieval* is independent from the o_{min} occurrences sent to the client, we do not report its unitary cost. We observe that the costs of the *Batched Text Retrieval* are more than halved w.r.t. the costs of the QueryNum procedure, which is mostly due to the smaller size of the outsourced array $\langle s \rangle$ w.r.t. the encrypted array $\langle A_P \rangle$, in turn leading to faster PIR queries. All the unitary costs show the expected linear and poly-logarithmic trends in server and client costs, respectively.

The overall computational costs of the privacy-preserving query for the well-formed star-free pattern $\mathfrak{p}$ are reported in Fig. 3.7(b). From the experimental data, we observe that the performance mostly depend on the costs for *Batched Text Retrieval*, which grow linearly with the size of the dataset. In case of the client cost, the linear trend is due to the increasing number o_{min} of portions of s that must be retrieved: indeed, the occurrences o_{min} of the substring $GCCTATCG$ (i.e., the wildcard-free one with the least number of occurrences) vary from 1 to 41 with an increasing dataset size. In case of the server cost, the linear trend is given by the growth of both the unitary

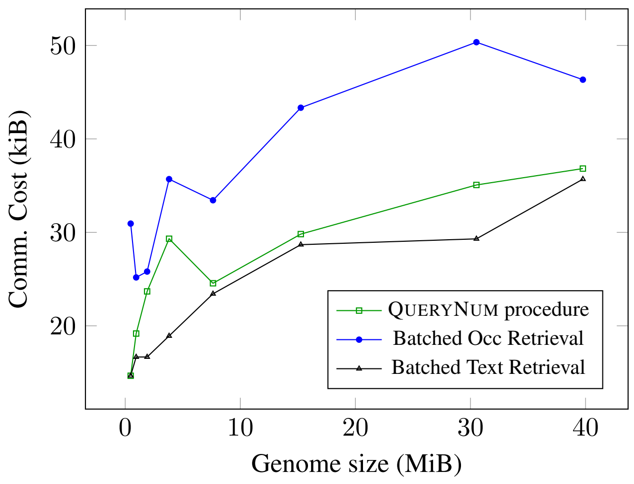


Figure 3.8: *Communication cost of the* STARFREEQUERY *procedure to retrieve all the occurrences of the well-formed star-free pattern* $\mathfrak{p} = ?(GC|A|)GCCTATCG(G|TAC|??)([!CT]?|)TA\ ?(TG|CGT|TA|[ACG][ATG])GTC(|?)$

cost and the number of retrievals, which grow linearly with the dataset size and o_{min}, respectively. The costs of the QUERYNUM and *Batched Occ Retrieval* are less affected by the increasing number of occurrences, showing similar trends to the corresponding operations performed in the evaluation of the QUERY procedure (Fig. 3.3).

Overall, we observe that the response time of our privacy-preserving pattern matching queries over the biggest dataset (i.e., the entire chromosome), given by the sum of the three components reported in Fig. 3.7(b), is about 35 minutes, which amounts to approximately four times the cost of the QUERY procedure for wildcard-free substrings. This performance gap is mostly due to the dependence of the server cost on the number of occurrences o_{min}, which is a non tight upper bound on the number of matches of $\mathfrak{p}$, while we recall that the server cost of the QUERY procedure, when the batched retrieval method is employed, is independent from the number of occurrences of the string. Nonetheless, we remark that the observed performance gap is acceptable, given the unprecedented expressiveness achieved by our privacy-preserving pattern matching queries.

In addition, although not reported in the client cost of our pattern matching query, we experimentally verified that identifying the occurrences of $\mathfrak{p}$ in the portions of the string s fetched from the outsourced array $\langle s \rangle$, which corresponds to lines 20– 24 in Alg.3.6, has a negligible impact on client cost: indeed, this operation requires 180 μs averaged over all the dataset sizes reported in Fig. 3.7, while the overall client cost is on average about 30 seconds, hereby showing 5 order of magnitudes of difference.

Concerning the communication cost of our query, reported in Fig. 3.8, we observe that it is roughly equivalent to the cost reported for the QUERY procedure in Fig. 3.3. Indeed, also in case of STARFREEQUERY procedure the communication cost is mostly due to the batched retrieval of the occurrences, as it fetches in a single round an amount of data proportional to the number of occurrences o_{min}. Aside from the non smooth behavior, which is given by the different number of recursive levels obtained by employing an optimal value of the radix b in the Lipmaa's PIR protocol, the overall communication cost still exhibits the expected polylogarithmic trend, showing that our PPSS protocol allows to increase the expressiveness of the queries while retaining approximately the same bandwidth.

Finally, we did not thoroughly evaluate the overall performance of the QUERYPATTERN procedure, as it is rather obvious from its description in Alg. 3.7 that the computational and communication costs can be easily derived from the corresponding costs of the k STARFREEQUERY procedures invoked at line 4. Conversely, we focused our evaluation on the estimation of the impact on the client cost of the QUERYPATTERN procedure of the MATCHOCC procedure invoked at line 8 in Alg. 3.7, which computes the occurrences of the well-formed pattern $p = \alpha_1 * \cdots * \alpha_k$ from the occurrences of the k well-formed star-free patterns returned by the k STARFREEQUERY procedures. In our evaluation, we employed the well-formed pattern $\mathfrak{p} = GCAATC * CTGAC * TGAC$, as we considered it as a good representative of the well-formed patterns that may be searched by users of our PPSS protocol; indeed, since in general the $*$ wildcard does not significantly restrict the number of matches of the well-formed star-free patterns composing the matched well-formed pattern, we expect that users in our PPSS protocol would issue queries for patterns composed by infrequent well-formed star-free patterns, in order to avoid an unnecessary blowup of the number of matched occurrences. Our evaluation revealed that the impact of MATCHOCC procedure is limited: indeed, it requires only 3 s to compute the 7459 occurrences of the pattern $\mathfrak{p}$ over the entire chromosome, which is the biggest of our datasets, from the occurrences of the patterns *GCAATC*, *CTGAC*, *TGA*, as opposed to the 71.5 s of overall client cost of the three STARFREEQUERY procedures invoked to compute the occurrences of the patterns *GCAATC*, *CTGAC*, *TGA*.

CHAPTER 4

ObSQRE: Privacy-Preserving Substring Search Protocol Based on Intel SGX

In this chapter, we describe ObSQRE, our PPSS protocol with optimal communication cost based on Intel SGX technology. ObSQRE makes the information leakage coming from SGX side channels useless for the adversary by employing a substring search algorithm with data-independent control flow based on the backward search method (see Sec. 1.4), and by relying on the privacy guarantees of a DORAM protocol to obliviously access its data structures. We structure the description of ObSQRE as follows. First, we show how to build our PPSS protocol, relying on two components: an oblivious substring search algorithm and a DORAM; then, we describe the design of these components, providing our own doubly oblivious versions of Path, Ring and Circuit ORAMs (Sec. 4.1), and two oblivious substring search algorithms based on the backward search method (Sec. 4.2).

ObSQRE Overview

ObSQRE is a single-user PPSS protocol involving two entities: a data owner with limited computational capabilities and an untrusted server equipped with Intel SGX technology. We sketch the architecture of ObSQRE in Fig. 4.1. ObSQRE is composed by two main modules: a simple application run at client side, which communicates with the enclave to provide the data needed for substring search queries and to receive back the results; an SGX based application, which actually executes substring search queries inside an SGX enclave hosted on an untrusted server. The SGX application provides the two main components of ObSQRE: an oblivious substring search algorithm that employs the outsourced full-text index to efficiently perform substring search queries; the client algorithms of a DORAM, which is employed to obliviously retrieve entries

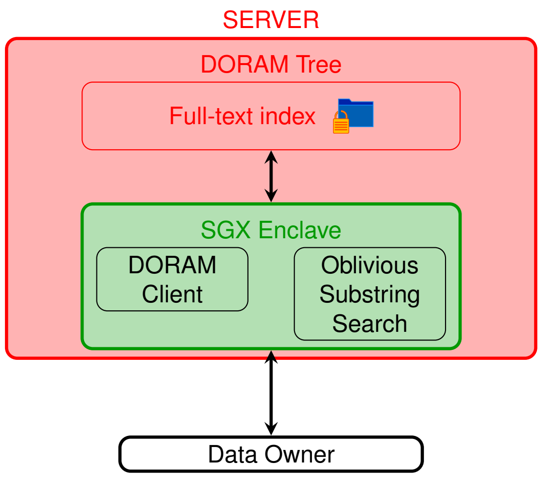


Figure 4.1: *Overview of the ObSQRE PPSS architecture*

from the full-text index.

ObSQRE is composed by three procedures: SETUP, LOAD and QUERY. The first one, executed by the data owner once as a pre-processing stage, computes the full-text index from the document collection $\mathbf{D}$. The index is encrypted with a semantically secure symmetric cipher and outsourced with the encrypted documents to the untrusted server. The integrity of the index is guaranteed by encrypting it with an Authenticated Encryption with Authenticated Data (AEAD) scheme, such as Galois Counter Mode (GCM), whose encryption procedure computes also a keyed digest that is verified upon decryption. Whenever the data owner is willing to perform queries, it asks the untrusted server to instantiate the ObSQRE SGX enclave, employing the remote attestation procedure, provided by SGX technology, to verify that the enclave has been correctly instantiated with the proper code and data by the remote server and to establish a secure communication channel with the enclave. When enclave is running, the LOAD procedure instantiates the DORAM data structures (both inside and outside the enclave). Then, the enclave receives via secure channel the decryption key and the digest for the full-text index from the data owner, decrypts the index, hereby verifying also its integrity, and stores it in the DORAM. Once LOAD is over, the data owner can submit queries to the enclave. In the QUERY procedure, the data owner sends through the secure channel the substring q to be searched. Then, the oblivious substring search algorithm runs inside the enclave, obliviously accessing the full-text index to compute the positions of the occurrences of q over documents of $\mathbf{D}$, which are sent back to the data owner.

The privacy guarantees of ObSQRE follows from the security guarantees of SGX applications and from the obliviousness of the algorithms run inside the enclave. Thus, the main technical challenge in the design of ObSQRE lies in devising efficient constructions for the two main components of our solution, namely a DORAM protocol and an oblivious substring search algorithm. In the following, we present the design of three DORAMs, which are doubly oblivious versions of the ORAMs described in Sec. 1.6, and two oblivious substring search algorithms based on the backward search method, which are differentiated by the adoption of distinct oblivious algorithms for the RANK procedure.

In all our constructions, we rely on these two simple operations: oblivious write OBLWRITE and oblivious swap OBLSWAP. The former (resp. the latter), given three input parameters $cond$, a and b, writes the content of b to a (resp. swaps the content of b and a) if and only the boolean expression $cond$ is true. To implement OBLWRITE, we employ the `x86_64` assembly instruction `CMOVNZ`, which moves the content of the source operand to the destination one if the zero flag is not set. `CMOVNZ` is oblivious as its operands are always loaded in the CPU and written back regardless of the status of the flag. The OBLSWAP operation, given the input parameters $cond, a, b$, first computes OBLWRITE($cond$, tmp, $a \oplus b$), where tmp is initially set to 0; then, it updates a and b with $a \oplus tmp$ and $b \oplus tmp$, respectively.

4.1 Doubly Oblivious RAMs

In ObSQRE, the ORAM client is executed inside the SGX enclave, therefore its memory access pattern is leaked through SGX side channels. This information is sufficient to invalidate the privacy guarantees of the ORAMs described in Sec. 1.6: for instance, if the adversary can observe which block is moved to the stash by the FINDBLOCK procedure in Ring and Circuit ORAMs, it can immediately learn the block requested by the client. Therefore, to prevent this leakage, we make the client algorithms of the three ORAMs described in Sec. 1.6 oblivious, obtaining three corresponding DORAMs. We start our description with required modifications to the stash and the recursive position map which apply to all our three DORAMs.

In all our DORAMs, the stash cannot have a dynamic size, lest the number of blocks moved between the DORAM and the stash is leaked. Thus, in all our DORAMs the stash has a fixed size S, and an *overflow* error occurs if the number of blocks in the stash is higher than S. Empty entries in the stash are filled with dummy blocks. The stash analysis of Path, Ring and Circuit ORAMs provide upper bounds for S making the probability of overflows negligible. We observe that all these upper bounds are independent from the number of blocks stored in the ORAM, thus the stash size S can be neglected while evaluating the asymptotic complexity of our DORAMs.

If a recursive position map is employed, the ORAMs that store the position map must be doubly oblivious too. Once a block from each of these DORAMs is fetched, the client obliviously swaps each one of the C entries in the block with a memory location `dest`, initialized with the new leaf id lid', actually performing the swap only for the entry corresponding to the block to be retrieved from the next DORAM in the recursion. Eventually, `dest` stores the id of the path to be fetched from the next DORAM, while the corresponding entry in the block stores the updated id lid'.

In addition, we enrich all our DORAMs with a mechanism to efficiently detect any tampering (including replacement with old blocks) on any path fetched from the DORAM, while storing in the enclave a single digest for the whole DORAM. Specifically, we combine the DORAM tree with a Merkle tree, as proposed in [127] for Path ORAM; however, we encrypt the buckets with an AEAD scheme to replace an unkeyed hash computation for the digest of each bucket with a more efficient symmetric encryption operation.

Path DORAM

We start with a description of the oblivious EVICTION procedure proposed in the Path DORAM of ZeroTrace [135] and employed with minor modifications in all existing works [4, 111]; then, we show the improvements introduced in our eviction strategy. The EVICTION procedure of ZeroTrace, for each block of the stash (even dummy ones), sweeps over the evicted path, which is initialized with dummy blocks, from the leaf to the root bucket, obliviously swapping each block with the entry of the stash at hand; the block of the stash is actually swapped with a dummy block found in the deepest non-full bucket that can store the block at hand. The computational cost of ZeroTrace EVICTION is thus $\mathrm{O}((S+\log(M)\cdot Z)\log(M)\cdot Z\cdot B)$, since both the blocks of the stash and the blocks of the path fetched by the FINDBLOCK procedure, which are appended to the stash too, must be obliviously swapped with all the blocks in the evicted path.

In our Path DORAM, we modify this EVICTION procedure by introducing an optimization, called *in-place eviction*, that allows to approximately halve its cost. Indeed, instead of appending all the blocks of the evicted path to the stash and then evict them as all other blocks in the stash, the client tries to push these blocks as down as possible in the path before performing stash eviction. This optimization allows to swap a block in the path only with deeper buckets instead of swapping it with all the buckets in the path. We remark that this optimization has no impact on the overflow probability of the DORAM, as evictions always aim at moving blocks as down as possible in the path, leaving free slots in higher buckets since they can store a large amount of the DORAM blocks. Even with this optimization, our oblivious EVICTION still exhibits an $\mathrm{O}(\log(M)\cdot Z)$ computational overhead with respect to the non oblivious EVICTION of Path ORAM. To reduce the impact of this overhead over DORAM accesses, in our Path DORAM we aim at making evictions less frequent. To make evictions unnecessary after each FINDBLOCK, as in Ring ORAM, we need to ensure that only one block is appended to the stash for each access. We achieve this employing the FINDBLOCK procedure of Circuit ORAM, which actually moves to the stash only the block bid, if found in the fetched path, and writes back the fetched path to the tree, replacing the block bid with a dummy one. We note that, although we add a write back operation for each access, this strategy allows our Path DORAM to perform evictions every $A>1$ accesses; since a write back costs $\mathrm{O}(\log(M)\cdot Z\cdot B)$, it is asymptotically faster than an eviction, thus improving the performance of our Path DORAM. To choose which path to evict, our Path DORAM employs the same deterministic schedule of Ring ORAM; hence, as the stash is managed in the same fashion, the values of Z, S and A suggested by authors of Ring ORAM [128] can be employed in our Path DORAM too.

Our oblivious FINDBLOCK procedure, reported in Alg. 4.1, starts by fetching the path with id lid from the DORAM tree (line 1) and then it obliviously looks for the block with id bid over the fetched path (lines 2-3). If the block is found, it is moved to a variable `dest` and replaced by a dummy block in the fetched path (line 3), otherwise neither `dest` nor the fetched path are updated. Then, the FINDBLOCK procedure obliviously sweeps (lines 4-9) over the stash to either write to the stash the block bid, if found in the fetched path, or to search the block in the stash. In the former case, the proper update of `write` flag (line 9) ensures that the block bid is written to the first empty entry in the stash (line 7); in the latter case, the block bid found in the stash is written to `dest` (line 6). In both cases, the leaf id of the block bid in the stash is

Algorithm 4.1: FINDBLOCK in Path/Circuit DORAMs

```
Input: $bid$: id of the block to be retrieved from the DORAM
       $lid$: id of the path where block $bid$ may be located
       $lid'$: id of the path where block $bid$ will be evicted
Output: The block with id $bid$
Data: Stash: $S$ blocks not evicted to the DORAM yet
1  Blks $\leftarrow$ READPATH($lid$) , dest.bid $\leftarrow \perp$
2  foreach blk $\in$ Blks do
3      OBLSWAP(blk.bid $= bid$, dest, blk)
4  write $\leftarrow$ dest.bid $= bid$
5  foreach blk $\in$ Stash do
6      OBLWRITE($bid =$ blk.bid, dest, blk)
7      OBLWRITE(blk.bid $= \perp \wedge$ write, blk, dest)
8      OBLWRITE($bid =$ blk.bid, blk.lid, $lid'$)
9      write $\leftarrow$ write $\wedge$ blk.bid $\neq bid$
10 WRITEPATH(Blks, $lid$)
11 return dest
```

Table 4.1: *Format of the bucket metadata in Ring DORAM. Grayed-out fields must necessarily be encrypted. M is the number of blocks in the ORAM, while Z (resp. D) denotes the maximum (resp. minimum) number of real (resp. dummy) blocks per bucket*

Field	Bit width	Size	Description
IV	λ	1	IV for bucket encryption
Bids	$\log(M{+}1)$	$Z{+}D$	Block ids of all blocks
Lids	$\log(M{+}1)$	Z	Leaf ids of real blocks
Invalid	1	$Z{+}D$	Flags keeping track of invalid blocks
Cnt	$\log D$	1	Count accesses to bucket

updated to lid' (line 8). Finally, the FINDBLOCK procedure writes back the fetched path to the DORAM tree (line 10). The computational cost of our oblivious FINDBLOCK is mostly due to the linear sweeps over the fetched path and the stash, whose costs amount to $O(\log(M){\cdot}Z{\cdot}B)$ and $O(S{\cdot}B)$, respectively; therefore, our oblivious FINDBLOCK procedure exhibits the same cost of the non oblivious version, namely $O(\log(M){\cdot}Z{\cdot}B)$.

Ring DORAM

We start the description of our Ring DORAM by discussing the structure of the metadata of the buckets, reported in Tab. 4.1, which differs from the one of buckets in the original Ring ORAM. We recall that Ring ORAM enriches each bucket with metadata that are employed by the SELECTOFFSET procedure to choose, for each bucket, one block to be retrieved from the ORAM tree. We observe that the major difference between the metadata employed in our Ring DORAM and the ones employed in Ring ORAM, which are reported in Tab. 1.2, is the absence of the permutation Π that stores the offsets of the real blocks in the bucket. Indeed, in our Ring DORAM we decide to store also the block ids of the dummy blocks in the bucket, in turn making the information found in Π redundant. This solution, although slightly increasing the size of the bucket metadata (we save $Z\log(Z{+}D)$ bits for Π, but we add $D\log(M)$ bits for the block ids of dummy blocks), allows to obliviously compute with a single sweep over

Algorithm 4.2: SELECTOFFSET in Ring DORAM

```
Input: $bid$: id of the block to be fetched from the ORAM
       Meta: metadata of a bucket of the path where block $bid$ may reside
Output: off: position in the bucket of the block with id $bid$, if found, otherwise the position of a
        randomly chosen valid dummy
1 found $\leftarrow$ false, max $\leftarrow -1$
2 for i $\in \{0, \dots, Z + D - 1\}$ do
3     if $\neg$Meta.invalid[i] then
4         rnd $\xleftarrow{R} \{0, \dots, 255\}$
5         max_dummy $\leftarrow$ Meta.bids[i]$=\perp \wedge$ rnd $>$ max
6         sel $\leftarrow$ Meta.bids[i]$=bid \vee$ (max_dummy $\wedge \neg$found)
7         OBLWRITE(sel, off, i), OBLWRITE(max_dummy, max, rnd)
8         found $\leftarrow$ found $\vee$ Meta.bids[i] $= bid$
9 return off
```

the metadata the offset of the chosen block in the SELECTOFFSET procedure.

We now describe how to make the procedures of Ring ORAM oblivious, starting with the SELECTOFFSET one, which is reported in Alg. 4.2. This procedure iterates over all the blocks in the bucket, skipping invalid ones (line 3) as they cannot be chosen. Note that there is no need to hide which blocks are skipped, as the adversary can easily know which blocks are invalid by logging blocks chosen in previous accesses to the bucket at hand. For each valid block, a number rnd is uniformly sampled from a fixed domain (e.g., $\{0, \dots, 255\}$ in line 4) and the offset of the valid block is obliviously written to the variable off (line 7). The update of max_dummy (line 5), sel (line 6) and found (line 8) flags ensures that eventually the variable off stores either the position in the bucket of the block with id bid, if found in the bucket, or the position of the valid dummy block with the highest random number among the ones sampled for all the dummy valid blocks, otherwise. As all these numbers are sampled from the same distribution, each one of them has the same probability of being the highest, thus this method chooses uniformly at random a block among the dummy valid ones without revealing which blocks are dummies.

In the oblivious EARLYRESHUFFLE procedure, Z blocks have to be randomly placed over $Z{+}D$ slots of a bucket. To this end, the i-th block, $1{\leq}i{\leq}Z$, is obliviously written in the off-th free slot of the bucket, where off is uniformly sampled from $\{1, \dots, Z{+}D{-}i{+}1\}$; the block and leaf ids in the bucket metadata are updated accordingly. Since the bucket is initialized with all dummy blocks, after Z sweeps, each writing one block, D slots of the bucket certainly contain a dummy block. As this strategy requires Z linear sweeps over a bucket with $Z{+}D$ slots, EARLYRESHUFFLE costs $\mathrm{O}(Z{\cdot}(Z{+}D){\cdot}B)$ per bucket. We show in Appendix 4 that this strategy ensures that each block is placed with uniform probability over all the $Z{+}D$ slots of the bucket.

In the oblivious FINDBLOCK procedure, reported in Alg. 4.3, first the metadata for all the buckets along the path with id lid are fetched (line 1). Then, the procedure iterates over the metadata to choose one block per bucket to be retrieved from the server (lines 2-5). The offset of the chosen block in the bucket, obliviously computed by the SELECTOFFSET procedure (line 3), is appended to the set Offsets (line 4). Furthermore, the bucket metadata are updated by marking the chosen block as invalid and by increasing the number of accesses to the bucket (line 5). Subsequently, the algo-

Algorithm 4.3: FINDBLOCK in Ring DORAM

```
Input: $bid$: id of the block to be retrieved from the DORAM
       $lid$: id of the path where block $bid$ may be located
       $lid'$: id of the path where block $bid$ will be evicted
Output: the block with id $bid$
Data: Stash: $S$ real/dummy blocks not evicted to the DORAM yet
1  Metadata $\leftarrow$ FETCHBUCKETSMETADATA($lid$), Offsets $\leftarrow \emptyset$
2  foreach Meta $\in$ Metadata do
3      off $\leftarrow$ SELECTOFFSET($bid$, Meta)
4      Offsets $\leftarrow$ Offsets $\cup$ {off}
5      Meta.invalid[off] $\leftarrow$ true, Meta.cnt ++
6  Blks $\leftarrow$ FETCHBLOCKS($lid$, Offsets), dest.bid $\leftarrow \perp$
7  foreach blk $\in$ Blks do
8      OBLSWAP(blk.bid $= bid$, dest, blk)
9  write $\leftarrow$ dest.bid $= bid$
10 foreach blk $\in$ Stash do
11     OBLWRITE($bid =$ blk.bid, dest, blk)
12     OBLWRITE(blk.bid $= \perp \wedge$ write, blk, dest)
13     OBLWRITE($bid =$ blk.bid, blk.lid, $lid'$)
14     write $\leftarrow$ write $\wedge$ blk.bid $\neq bid$
15 foreach i $\leftarrow 0$ to $\log(\frac{M+1}{2})-1$ do
16     if Metadata[i].cnt $\geq D$ then
17         Blks $\leftarrow$ FETCHVALIDBLOCKSINBUCKET($lid$, i)
18         Bucket $\leftarrow$ EARLYRESHUFFLE(Blks, Metadata[i])
19         WRITEBUCKET($lid$, i, bucket)
20 WRITEMETADATA($lid$, Metadata)
21 return dest
```

rithm employs the offsets found in `Offsets` to fetch from the path lid in the DORAM tree the $\mathrm{O}(\log(M))$ blocks chosen by the SELECTOFFSET procedure (line 6). Then, the client obliviously searches the block with id bid through a linear sweep over the fetched blocks (lines 7-8). The algorithm proceeds by iterating over the stash (lines 9-14) to either insert the block retrieved from the server (line 12) or locate the block with id bid in the stash, in case it was not found in the path with id lid (line 11). Afterwards, the FINDBLOCK procedure, for each bucket, invokes, if necessary (line 16), the EARLYRESHUFFLE procedure (line 18), fetching the Z valid blocks left in the bucket from the DORAM tree (line 17) and writing the whole bucket back after the reshuffle (line 19). Lastly, the FINDBLOCK procedure writes back the updated metadata to the DORAM tree (line 20). The computation cost of the FINDBLOCK procedure is mostly due to the linear sweep over the fetched blocks and the stash, as we neglect the cost of processing the bucket metadata because of their small size compared to a whole bucket. Therefore, our oblivious FINDBLOCK procedure exhibits the same cost of the non oblivious version, namely $\mathrm{O}(\log(M) \cdot B)$.

The EVICTION procedure follows the blueprint of the oblivious one described in Path DORAM. Indeed, although buckets in Ring DORAM has $Z+D$ slots, at most Z of them may be filled with real blocks; hence, buckets with Z slots can be employed during evictions, as in Path DORAM. At the end of the EVICTION, the EARLYRESHUFFLE procedure is invoked to randomly place each of the Z evicted blocks in a bucket with $Z+D$ slots, which is written back to the DORAM tree after re-encryption. As in Path DORAM, EVICTION is split in two phases: in-place and stash eviction. While the latter works exactly as in Path DORAM, in the former, for each bucket, only the $R \leq Z$

Algorithm 4.4: EVICTION in Circuit DORAM

```
Input: Path: path to be evicted
       $lid$: id of Path
Data: Stash: $S$ real/dummy blocks not evicted to the DORAM yet
1  dest $\leftarrow$ COMPUTEDESTINATIONS(Path, $lid$)
2  max_depth $\leftarrow -1$, target $\leftarrow$ dest[0], hold.bid $= \perp$
3  foreach blk $\in$ Stash do
4      depth $\leftarrow$ MAXDEPTH(blk.lid, $lid$)
5      OBLSWAP(dest[0]$\neq\perp\wedge$depth$>$max_depth, hold, blk)
6      OBLWRITE(depth $>$ max_depth, max_depth, depth)
7  for i $\leftarrow 0$ to $\log(N{+}1){-}2$ do
8      max_depth $\leftarrow$ i, deeper_bucket $\leftarrow$ (target$\neq$i $\wedge$ target$\neq\perp$)
9      foreach blk $\in$ Path[i] do
10         depth $\leftarrow$ MAXDEPTH(blk.lid, $lid$)
11         swap $\leftarrow$ (dest[i+1]$\neq\perp$ $\wedge$ depth$>$max_depth) $\vee$ (dest[i+1]$=\perp\wedge$blk.bid$=\perp$)
12         OBLSWAP(swap $\wedge$ $\neg$deeper_bucket, hold, blk)
13         OBLWRITE(depth$>$max_depth, max_depth, depth)
14     OBLWRITE($\neg$deeper_bucket, target, dest[i+1])
```

valid real blocks must be evicted, while all the other $Z{+}D{-}R$ blocks can be discarded, as they are either dummies or invalid real blocks. To avoid leaking R to the adversary, during in-place eviction the client has to always choose Z blocks for each bucket. In particular, for each bucket, the client must choose Z blocks out of the $V{\geq}Z$ valid blocks: R real valid blocks and $Z{-}R$ blocks among the $V{-}R$ dummy valid blocks. To this extent, we employ the Knuth's algorithm reported in [88, pag. 142], which chooses uniformly at random k elements out of h ones, with $k{\leq}h$; this algorithm can be trivially made oblivious by relying on oblivious write/swap primitives while retaining $\mathrm{O}(h)$ computational complexity. Once the offsets of the Z blocks are computed, they are obliviously fetched from the bucket with Z linear sweeps and evicted in deeper buckets. The computational cost of in-place eviction amounts to $\mathrm{O}(\log(M){\cdot}Z(\log(M){\cdot}Z{\cdot}B + (Z{+}D)B)) = \mathrm{O}(\log^2(M){\cdot}Z^2{\cdot}B)$, as $\log(M){\cdot}Z$ blocks must be fetched from their bucket with a linear sweep, which costs $\mathrm{O}((Z{+}D)B)$, and obliviously swapped with all the blocks found in deeper buckets. Therefore, the overall computational cost of EVICTION procedure is $\mathrm{O}(\log(M){\cdot}B{\cdot}Z(\log(M){\cdot}Z{+}S{+}Z{+}D)) = \mathrm{O}(\log^2(M){\cdot}Z^2{\cdot}B)$, which is the sum of the costs of in-place eviction, stash eviction and EARLYRESHUFFLE for all the buckets along the evicted path, respectively; this cost is amortized over $A{\geq}1$ DORAM accesses.

Circuit DORAM

The simplicity of the client in Circuit ORAM makes its oblivious design the easiest one among our three DORAMs. The FINDBLOCK procedure in our Circuit DORAM is equivalent to our Path DORAM, reported in Alg. 4.1. Conversely, the EVICTION procedure of Circuit ORAM significantly differs from the one of Path and Ring ORAMs. Specifically, the non-oblivious eviction involves two sweeps over the metadata of the evicted path and a single sweep over the evicted path. In the oblivious EVICTION procedure, reported in Alg. 4.4, the two sweeps over the metadata are performed by the COMPUTEDESTINATIONS procedure (line 1), which computes, for the stash and for each bucket in the evicted path, the additional metadata `dest`. We recall that these meta-

Table 4.2: *Client-side asymptotic computational costs*

	FINDBLOCK		EVICTION	
	ORAM	**our DORAM**	**ORAM**	**our DORAM**
Path	$\mathrm{O}(\log(N)\cdot Z\cdot B)$	$\mathrm{O}(\log(N)\cdot Z\cdot B)$	$\mathrm{O}(B\cdot\log(N)\cdot Z)$	$\mathrm{O}(\frac{\log^2(N)\cdot Z^2\cdot B}{A})$
Ring	$\mathrm{O}(\log(N)\cdot B)$	$\mathrm{O}(\log(N)\cdot B)$	$\mathrm{O}(\frac{B\cdot\log(N)\cdot Z}{A})$	$\mathrm{O}(\frac{\log^2(N)\cdot Z^2\cdot B}{A})$
Circuit	$\mathrm{O}(\log(N)\cdot Z\cdot B)$	$\mathrm{O}(\log(N)\cdot Z\cdot B)$	$\mathrm{O}(\log(N)\cdot Z\cdot B)$	$\mathrm{O}(\log(N)\cdot Z\cdot B)$

data specify how the blocks must be moved among buckets in the subsequent sweep over the evicted path (see Sec. 1.6); specifically, for each $i \in \{0,\dots,\log(M+1)-1\}$ (`dest`$[0]$ refers to the stash), `dest`$[i]$ stores the bucket where the block of the i-th bucket that can go deepest in the path must be moved, while `dest`$[i]=\perp$ if no block from the the i-th bucket must be moved down in the path. To avoid leaking these metadata while computing them, COMPUTEDESTINATIONS procedure employs oblivious writes to remove conditional dependent updates to these metadata. During the sweep over the evicted path (lines 2-14), at most one block, stored in `hold`, is simultaneously moved down along this path; the variable `target` stores the destination bucket of such block. Throughout the sweep over the evicted path, a procedure MAXDEPTH is employed to compute the deepest bucket of the path that can store a given block by hinging upon the leaf id of the block at hand and the id of the evicted path. First, the block in the stash that can go deepest in the path is obliviously moved to `hold` through a linear sweep of the stash (lines 3-6). Then, this block is moved to its destination bucket, where it is swapped (line 12) with either a dummy block, in case no block in the destination bucket must be moved down (i.e., if `dest`$[i+1]=\perp$ in line 11), or with the block in the destination bucket than can go deepest in the path (lines 11, 13). The computational cost of oblivious EVICTION is $\mathrm{O}(B\cdot(S+\log(M)\cdot Z)) = \mathrm{O}(B\cdot\log(M)\cdot Z)$, given by the linear sweeps over the stash and the evicted path, respectively.

To conclude the description of our DORAMs, we summarize the computational costs of their oblivious client algorithms in Tab. 4.2. We observe that Circuit DORAM is asymptotically faster than Ring and Path DORAMs; nonetheless, as 3 paths have to be fetched and written back for each DORAM access, a performance gain may be observed only for DORAMs with a significant number of blocks. Instead, Ring DORAM saves a factor of Z in the computational cost of the FINDBLOCK procedure; nonetheless, its oblivious algorithms involve cumbersome operations, which may increase the actual response time of DORAM accesses.

4.2 Oblivious Substring Search Algorithms

We now present two oblivious substring search algorithms based on the backward search method, which are safely executed inside the SGX enclave. These algorithms differ in the strategy employed to compute the RANK procedure (see Def. 1.7), which is a fundamental building block of backward search. We first describe these two oblivious algorithm for the RANK procedure, and then we show how to obtain an oblivious backward search algorithm by hinging upon either of these procedures.

Oblivious RANK with Augmented BWT Full-Text Index

Similarly to our multi-user PPSS protocol described in Chapter 3, this algorithm employs the ABWT full-text index to compute the RANK procedure. We recall that, for a string $s \in \Sigma^n$, the ABWT A_P, with sample period P, is an array with $\lceil \frac{n+1}{P} \rceil$ entries, each containing a pair of elements $(rank, l)$; for the i-th entry of A_P, $A_P[i].rank$ is a dictionary of $|\Sigma|$ entries that binds to a character $c{\in}\Sigma$ the value $\textsc{Rank}(c, i{\cdot}P{-}1)$, while $A_P[i].l$ is a string of P characters, namely the substring $L[i{\cdot}P, \ldots, (i{+}1){\cdot}P{-}1]$ of the BWT L. The value $\textsc{Rank}(c, i)$, $c{\in}\Sigma$, $i{\in}\{0, \ldots, n\}$, is computed from the $j{=}\lfloor \frac{i}{P} \rfloor$-th entry of A_P as the sum of $A_P[j].rank[c]$ and the number of occurrences of character c in $A_P[j].l[0, \ldots, i \bmod P]$. In the oblivious implementation of RANK procedure, reported

Algorithm 4.5: Oblivious RANK procedure based on ABWT for a string $s \in \Sigma^n$ with BWT L

```
Input: c: character of the alphabet $\Sigma$
       i: integer in $\{0, \ldots, n{+}1\}$
Output: ctr: number of occurrences of $c$ in $L[0, \ldots, i]$
Data: DORAM: DORAM storing the ABWT $A_P$ with sample period $P$
1 Entry ← DORAM.ACCESS($\lfloor \frac{i}{P} \rfloor$)
2 foreach char ∈ $\Sigma$ do
3     OBLWRITE(c = char, ctr, Entry.rank[char])
4 for j ← 0 to $P - 1$ do
5     OBLWRITE(Entry.l[j] = c ∧ j ≤ i mod $P$, ctr, ctr + 1)
6 return ctr
```

in Alg. 4.5, the ABWT is stored inside a DORAM; the algorithm first fetches the block storing the $h{=}\lfloor \frac{i}{P} \rfloor$-th entry of A_P (line 1); then, `ctr` is obliviously set to $A_P[h].rank[c]$ through a linear sweep over the entries of $A_P[h].rank$ (lines 2-3); lastly, the algorithm sweeps over the string $A_P[h].l$, obliviously increasing by 1 `ctr` whenever a character among the first $i \bmod P{+}1$ ones equals c (lines 4-5). If a Circuit DORAM is employed to store the ABWT A_P, each access to an entry of A_P costs $\mathrm{O}(C{\cdot}\log^2(n){\cdot}Z{\cdot}B)$, where C is the recursive factor employed to build the recursive position map of the DORAM; this cost becomes $\mathrm{O}(C{\cdot}\log^3(n){\cdot}Z^2{\cdot}B)$ in case Path or Ring DORAMs are employed to store A_P, as each EVICTION costs $\mathrm{O}(\log^2(n){\cdot}Z^2{\cdot}B)$ instead of $\mathrm{O}(\log(n){\cdot}Z{\cdot}B)$. Given that $B = \mathrm{O}(|\Sigma|\log(n){+}P\log(|\Sigma|))$ (i.e., the size of an entry of the ABWT A_P) and Z, C, $|\Sigma|$ and P are small constants, our ABWT based oblivious RANK procedure has $\mathrm{O}(\log^3(n))$ computational cost when Circuit DORAM is employed to store the ABWT A_P.

Oblivious RANK with Binary Search BWT Full-Text Index

In the Binary Search BWT (BSBWT) method, instead of the ABWT array A_P, we employ a balanced BST as a full-text index constructed from the BWT L of a string $s \in \Sigma^n$. To build this index, we employ an enumeration `Enum` of characters $c{\in}\Sigma$. For each $j{\in}\{0, \ldots, n\}$, we create a node in the BST as a key-value pair $(\mathtt{Enum}(s[j]){\cdot}(n{+}1) + pos_L(s[j]), \textsc{Rank}(s[j], pos_L(s[j])))$, where $pos_L(s[j])$ denotes the position in the BWT L of the character $s[j]$. Once the BST is built, $\textsc{Rank}(c, i)$, $c{\in}\Sigma$, $i{\in}\{0, \ldots, n\}$, can be computed by looking-up the node with key $\mathtt{k}{=}\mathtt{Enum}(c){\cdot}(n+1){+}i$ in the balanced BST, as outlined in Alg. 4.6. If the node is found (line 3), then $\textsc{Rank}(c, i)$ equals the value stored in this node by construction of the BST (line 4). Otherwise, the last node ex-

Algorithm 4.6: Non-oblivious RANK procedure with balanced BST for a string $s \in \Sigma^n$ with BWT L

```
Input: c: character of the alphabet $\Sigma$
       i: integer in $\{0, \dots, n+1\}$
Output: RANK(c, i): number of occurrences of c in $L[0, \dots, i]$
Data: BST: balanced Binary Search Tree (BST) constructed from the string $s$
      Enum: enumeration of characters in $\Sigma$
      Occ: dictionary binding a char $c \in \Sigma$ to $\mathtt{Rank}_L(c, n)$
1  node ← BST.root, k ← Enum(c)·$(n+1)$+i
2  while node $\neq \perp$ do
3      if node.key = k then
4          return node.value
5      go_left ← 0, parent ← node, node ← node.right
6      if node.key < k then
7          go_left ← 1, node ← node.left
8  if parent.key<Enum(c)·$(n+1)$∨parent.key≥(Enum(c)+1)·$(n+1)$ then
9      return Occ(c)·go_left
10 return parent.value − go_left
```

plored is either the predecessor (if go_left$=0$) or the successor (if go_left$=1$) of the node with key k. Since Enum(c)$\cdot(n+1)$ is added to the key of each node in the construction of the BST, then all the nodes referring to occurrences of the same character c have consecutive keys; as a consequence, the predecessor node corresponds to the last occurrence of c in $L[0, \dots, i]$, while the successor node corresponds to first occurrence of c in $L[i+1, \dots, n]$. Thus, in case the look-up of a node with key k=Enum$(c)\cdot(n+1)+i$ ends up in the predecessor of such node, then RANK(c, i) equals the value of the predecessor node, while if the look-up ends up in the successor of the node with key k, then the value of the successor node must be decremented by 1 to obtain RANK(c, i) (line 10). In case there is no occurrence of c in $L[0, \dots, i]$ (resp. $L[i+1, \dots, n]$), the predecessor (resp. successor) node refers to an occurrence of a character $c' \neq c$, as checked in line 8; thus, RANK(c, i) equals 0 (resp. the number of occurrences of c in L), as returned in line 9.

To make this algorithm oblivious, each of the $\mathrm{O}(\log(n))$ nodes visited in the search path of the tree should be accessed with a DORAM. In particular, we rely on the Oblivious Data Structure (ODS) [157] framework to obliviously access nodes in the BST. ODS relies on the fact that each node of a BST can be reached only from another node, i.e., its parent in the tree, to store the position map entries inside ORAM blocks. Specifically, each node of the BST stores the ids of the paths of the ORAM tree containing the blocks that store the children of the node at hand. Therefore, to access any node of the BST, the client only needs to store the root node: from this one, the client chooses to fetch one of its two children, employing the corresponding leaf id stored in the root node. This procedure is repeated to visit the entire path of the BST. In this way, the look-up of a node in a BST with n nodes stored inside an ODS requires $\mathrm{O}(\log(n))$ *direct* (i.e., with no recursive position map) accesses, one for each level of the BST, to the DORAM. In our ODS, we roughly halve the look-up cost by applying a trick proposed by Gentry [62]: instead of storing all the nodes of the BST in the same DORAM, a distinct DORAM is employed for each level of the BST. In this way, the access to each level becomes faster, as it is performed over smaller DORAMs.

Employing distinct Circuit DORAMs (resp. Path or Ring DORAMs) to store each level of the BST allows to compute the oblivious RANK procedure with $\mathrm{O}(\log^2(n)\cdot Z\cdot B)$ (resp. $\mathrm{O}(\log^3(n)\cdot Z^2\cdot B)$) computational cost. Since $B{=}\mathrm{O}(\log(n))$ and $Z{=}\mathrm{O}(1)$, the BSBWT based oblivious RANK procedure has the same $\mathrm{O}(\log^3(n))$ computational cost of the ABWT one; nonetheless, the BSBWT method accesses $\log_2(n)$ DORAMs with small blocks instead of $\log_C(n)$ DORAMs with large blocks accessed in the ABWT method, in turn allowing different implementation tirade's.

Oblivious Backward Search

We now show how to obtain an oblivious substring search algorithm based on the backward search method, by hinging upon one of the two oblivious RANK procedures we have just described. First of all, we observe that the control flow of Alg. 1.2 depends only on the length m of the searched substring and on the number $o_q = \beta{-}\alpha$ of its occurrences. To avoid this dependence, we should always perform the same number of iterations in all the queries; thus, such number necessarily corresponds to an upper bound on the length of the searched substring and on its number of occurrences. Since a reasonable upper bound is at least proportional to n, then this strategy would introduce a significant performance slowdown; thus, to retain practical performance, in our PPSS protocol we decide to keep the control flow unmodified, at the cost of leaking both m and o_q to the adversary, deeming this information leakage as acceptable.

Therefore, to make the backward search method oblivious, we just need to make accesses to its data structures oblivious. In particular, our algorithm, besides employing an oblivious RANK procedure, must obliviously access both the dictionary `Count` and the SA Suf: indeed, the entries fetched from the former would leak the characters of q, while the entries retrieved from the SA would leak the values α and β, which are related to both the string s and the substring q. To prevent such information leakages, Suf is stored inside a DORAM, while `Count` (with its $|\Sigma|$ entries) is stored inside the enclave and each search over it is (obliviously) performed through a linear sweep. These implementation choices together with each of the proposed oblivious RANK procedures yield two substring search algorithms with computational cost $\mathrm{O}((m{+}o_q)\log^3(n))$.

4.3 Security Analysis of ObSQRE

We now discuss and formally define the security guarantees of ObSQRE, which are based on the confidentiality and integrity guarantees given by Intel SGX and on the access pattern privacy given by our DORAMs. Similarly to the security analysis of our multi-user PPSS protocol, reported in Sec. 3.3, we employ security definitions based on the simulation paradigm, which allows to precisely frame the information leakage $\mathcal{L}$ of a privacy-preserving protocol by showing that the execution of the protocol can be simulated by a simulator $\mathcal{S}$ employing all and only the information found in $\mathcal{L}$. Nonetheless, despite relying on the same framework employed in Sec. 3.3 for our multi-user PPSS protocol, we need to introduce a different security definition for ObSQRE, since we consider a different threat model. Indeed, while the security definition employed in our multi-user PPSS protocol (see Def. 3.6) withstands only semi-honest adversaries, who do not misbehave in the protocol but are willing to learn as much information as possible, ObSQRE is able to guarantee a limited information leakage and the correctness of the PPSS protocol even against malicious adversaries, who may arbitrarily deviate from the protocol specification in order to gain some additional information. More concretely, in our threat model we assume a powerful adversary that has total control of the machine hosting the SGX enclave. Thus, thanks to the security guarantees of SGX, the attacker has no direct access to the data stored inside the enclave and it cannot interfere with the computation performed within the enclave; nonetheless, it has access to every data outside the enclave and may tamper with the computation performed outside the enclave. Furthermore, the adversary can learn the memory access pattern of algorithms run inside the enclave through side channel attacks.

Since the security guarantees of ObSQRE are based on the privacy guarantees of our DORAMs, we perform our security analysis in two main steps: first, we define and prove the access pattern privacy guarantees of our DORAMs; then, we define and prove the security guarantees of ObSQRE assuming that a secure DORAM is employed.

DORAM Security Analysis

We assume that the DORAM stores a dataset D split in M blocks, each of size B bits. In our security definition, besides the ACCESS procedure that we have already described, we consider also an INIT procedure, which is employed to build the DORAM tree and insert the M blocks of D in the DORAM: specifically, for each block to be inserted, the algorithm obliviously adds to the stash the block at hand and then evicts the stash to the DORAM tree following the eviction strategy of the DORAM. In order to properly construct the DORAM tree, the INIT procedure employs several parameters specified by the user: the recursive factor C to build the recursive position map, the maximum number of real blocks per bucket Z, the stash size S, the number of dummy blocks per buckets D and the eviction period A. The security guarantees of a DORAM are not weakened if the adversary knows these values, since they depend only on the number of blocks of the DORAM; thus, to simplify our security analysis we assume that the INIT procedure always employs the same values for these parameters. Since our DORAMs employs an integrity-check mechanism based on Merkle-trees, both the INIT and ACCESS procedures may return the special value `abort` in case they detect data tampering on the DORAM.

As in the security analysis of our multi-user PPSS protocol, we specify the trace $\mathcal{T}$, that is the the information directly observed by the adversary while interacting with the DORAM, and the leakage $\mathcal{L}$, which corresponds to the information about the dataset and the accessed blocks that is inferred by the adversary from the trace $\mathcal{T}$.

Definition 4.1 (Trace and Leakage of DORAM). *Given a DORAM whose client runs inside an SGX enclave and whose DORAM tree is stored in the unprotected memory, the trace of the DORAM is $\mathcal{T} = \{Code_{AP}, Data_{AP}, Data_{Srv}\}$, with $Code_{AP}$ and $Data_{AP}$ denoting the code and data access patterns of the DORAM client, respectively, while $Data_{Srv}$ denotes the information sent by the DORAM client outside the enclave. The leakage $\mathcal{L}$ of the DORAM denotes the information inferred by the adversary from the trace $\mathcal{T}$ about the dataset D and the set of blocks accessed by the DORAM.*

In our security definition, we split the trace in two components $\mathcal{T}_{Init}$ and $\mathcal{T}_{Acc}$, which refer to the trace of the INIT and the ACCESS procedures, respectively. Similarly, we split the leakage $\mathcal{L}$ in two components $\mathcal{L}_{Init}$ and $\mathcal{L}_{Acc}$ that represent the information inferred by the adversary from $\mathcal{T}_{Init}$ and $\mathcal{T}_{Acc}$, respectively.

```
**Experiment** *transcript* $\leftarrow \mathbf{Real}_{\rho,\mathcal{A}}(\lambda)$:
$(D, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{Init}(1^{\lambda})$
$res_0 \leftarrow \rho.\textsc{Init}(\{D_i\}_{i=1}^{n})$
$\forall i \in \{1,\dots,d\}$:
    $(bid_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{Acc,i}(\mathtt{st}_{\mathcal{A}}, D, \{bid_j\}_{j=1}^{i-1}, \mathcal{L}_{Init}, \{\mathcal{L}_{Acc,j}\}_{j=1}^{i-1}, \mathcal{T}_{Init}, \{\mathcal{T}_{Acc,j}\}_{j=1}^{i-1})$
    $res_i \leftarrow \rho.\textsc{Access}(bid_i)$
$transcript \leftarrow \{\{res_i\}_{i=0}^{d}, \{\mathcal{T}_{Acc,i}\}_{i=1}^{d}, \mathcal{T}_{Init}, \mathtt{st}_{\mathcal{A}}\}$

**Experiment** *transcript* $\leftarrow \mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}(\lambda)$:
$(D, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{Init}(1^{\lambda})$
$(D^{\mathcal{S}}, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_{Init}(\mathcal{L}_{Init})$
$res_0 \leftarrow \rho.\textsc{Init}(\{D_i^{\mathcal{S}}\}_{i=1}^{n})$
$\forall i \in \{1,\dots,d\}$:
    $(bid_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_{Acc,i}(\mathtt{st}_{\mathcal{A}}, D, \{bid_j\}_{j=1}^{i-1}, \mathcal{L}_{Init}, \{\mathcal{L}_{Acc,j}\}_{j=1}^{i-1}, \mathcal{T}_{Init}, \{\mathcal{T}_{Acc,j}\}_{j=1}^{i-1})$
    $(bid_i^{\mathcal{S}}, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_{Acc,i}(\mathtt{st}_{\mathcal{S}}, \mathcal{L}_{Init}, \{\mathcal{L}_{Acc,j}\}_{j=1}^{i})$
    $res_i \leftarrow \rho.\textsc{Access}(bid_i^{\mathcal{S}})$
    **if** $res_i \neq \mathtt{abort}$: $res_i \leftarrow D_{bid_i}$
$transcript \leftarrow \{\{res_i\}_{i=0}^{d}, \{\mathcal{T}_{Acc,i}\}_{i=1}^{d}, \mathcal{T}_{Init}, \mathtt{st}_{\mathcal{A}}\}$
```

Figure 4.2: *Security experiments for DORAM protocol* ρ

Definition 4.2 (DORAM Security). *Given a security parameter* λ*, an integer* $d \geq 1$*, and a DORAM* ρ *with trace* $\mathcal{T} = \{\mathcal{T}_{Init}, \mathcal{T}_{Acc,1}, \dots, \mathcal{T}_{Acc,d}\}$ *and leakage* $\mathcal{L} = \{\mathcal{L}_{Init}, \mathcal{L}_{Acc,1}, \dots, \mathcal{L}_{Acc,d}\}$ *as in Def. 4.1, consider the two interactive experiments* $\mathbf{Real}_{\rho,\mathcal{A}}$ *and* $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$*, outlined in Fig. 4.2, between a challenger and an adversary* $\mathcal{A}$ *consisting of* $d{+}1$ *probabilistic polynomial time algorithms, i.e.,* $\mathcal{A}=\{\mathcal{A}_{Init}, \mathcal{A}_{Acc,1}, \dots, \mathcal{A}_{Acc,d}\}$*. Throughout the experiments, the challenger may invoke the DORAM* ρ *and a simulator* $\mathcal{S}$ *consisting of* $d{+}1$ *probabilistic polynomial time algorithms, i.e.,* $\mathcal{S} = \{\mathcal{S}_{Init}, \mathcal{S}_{Acc,1}, \dots, \mathcal{S}_{Acc,d}\}$*; the adversary* $\mathcal{A}$ *can tamper with data and computation of the DORAM as described in our threat model. Denoting as* $\mathcal{D}$ *a probabilistic polynomial time algorithm that, given the transcript* o *of an experiment, determines if* o *refers to* $\mathbf{Real}_{\rho,\mathcal{A}}$ *(*$\mathcal{D}(o) = 0$*) or* $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ *(*$\mathcal{D}(o) = 1$*) experiment, the DORAM* ρ*, with leakage* $\mathcal{L}$ *and trace* $\mathcal{T}$*, is secure against malicious probabilistic polynomial time adversaries* $\mathcal{A}$ *if, for every possible* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that for every* $\mathcal{D}$*:*

$$\mathtt{Pr}(\mathcal{D}(o) = 1 | o \leftarrow \mathbf{Real}_{\rho,\mathcal{A}}) - \mathtt{Pr}(\mathcal{D}(o) = 1 | o \leftarrow \mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}) \leq \epsilon(\lambda)$$

where $\epsilon(\cdot)$ *is a negligible function.*

In short, our security definition is satisfied if the transcripts of the two experiments outlined in Fig. 4.2 are computationally indistinguishable. This property is sufficient to prove these two security guarantees:

- The DORAM ρ leaks to any malicious adversary no more information than the leakage $\mathcal{L}$
- The DORAM client can detect any misbehavior of the malicious adversary that affects the result of the computation, in turn ensuring the correctness of the computation against any malicious adversary

The first guarantee stems from the indistinguishability between the trace $\mathcal{T}$ of the DORAM and the state $\mathtt{st}_{\mathcal{A}}$ of the adversary found in the transcripts of the two experiments.

Indeed, the trace of the DORAM $\mathcal{T}$ and the state $\texttt{st}_{\mathcal{A}}$ available to the adversary at the end of the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment depends on the fake input data $(D^{\mathcal{S}}, \{bid_i^{\mathcal{S}}\}_{i=1}^{d})$ constructed by the simulator $\mathcal{S}$. Since $\mathcal{S}$ constructs the fake input data by knowing only the leakage $\mathcal{L}$ provided by the challenger, no more information than $\mathcal{L}$ about the actual data can be inferred by the adversary from the operations observed over fake data. Since the trace $\mathcal{T}$ and the state $\texttt{st}_{\mathcal{A}}$ are computationally indistinguishable between the $\mathbf{Real}_{\rho,\mathcal{A}}$ and $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiments, then the trace $\mathcal{T}$ and the state $\texttt{st}_{\mathcal{A}}$ available to the adversary in the $\mathbf{Real}_{\rho,\mathcal{A}}$ experiment cannot reveal more information than $\mathcal{L}$ about the actual data.

The correctness of the computation stems from the indistinguishability between the outputs of the $d{+}1$ operations found in the transcripts of the two experiments. Indeed, in the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment, the challenger ensures that the result of the computation is always the correct one, unless the DORAM ρ has detected a misbehavior of the adversary. Thus, in case there exists a misbehavior of the adversary affecting the correctness of the result that is not detected by the DORAM, the result would be correct in the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment but wrong in the $\mathbf{Real}_{\rho,\mathcal{A}}$ one, hence making them distinguishable.

Our DORAMs provides the security guarantees given by Def. 4.2 for the information leakage $\mathcal{L}$ reported in the following statement:

Theorem 4.1. *Our DORAMs meet the security guarantees of Def. 4.2 against a malicious adversary with leakage $\mathcal{L} = \{\mathcal{L}_{Init}, \mathcal{L}_{Acc,1}, \ldots, \mathcal{L}_{Acc,d}\}$, where $\mathcal{L}_{Init} = \{M, B\}$ and $\mathcal{L}_{Acc,i} = \emptyset$, $i = 1, \ldots, d$.*

The proof of this theorem is found in Appendix 3.1. Informally, the privacy guarantees of our DORAMs stem from these two factors: the privacy guarantees of the original ORAMs schemes, which ensure that the access pattern observed over the DORAM tree reveals nothing about the blocks retrieved by the ACCESS procedure; the obliviousness of the client algorithms of our DORAMs, which is given by their control flow and memory access patterns being independent from the blocks retrieved by the ACCESS procedure. Furthermore, the correctness of the DORAM operations performed inside the enclave is ensured by the security guarantees of SGX, which prevent the adversary from tampering with any data or code stored in the enclave; the only operation performed outside the enclave is the fetch of the data from the DORAM tree, whose authenticity is ensured by the integrity-protection mechanism added to the DORAM tree.

ObSQRE Security Analysis

We now prove the security guarantees of ObSQRE, assuming that a DORAM fulfilling the security requirements of Thm. 4.1 is employed in our oblivious substring search algorithms. Some of these algorithms may employ several parameters, such as the sample period employed in the construction of the ABWT; since these parameters are not sensitive and can be derived by the adversary itself (e.g., from the block size of the DORAM storing the ABWT), for simplicity in our security analysis we assume that the same value is always employed. Similarly, we assume that the alphabet of the documents in the document collection $\mathbf{D}$ is publicly known.

```
Experiment $transcript \leftarrow \mathbf{Real}_{\mathcal{P},\mathcal{A}}(\lambda)$:
$(\mathbf{D}, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^\lambda)$
$\mathcal{I} \leftarrow \mathcal{P}.\textsc{Setup}(\mathbf{D})$
$res_0 \leftarrow \mathcal{P}.\textsc{Load}(\mathcal{I})$
$\forall i \in \{1, \dots, d\}$:
    $(q_i, occ_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i(\mathtt{st}_{\mathcal{A}}, \mathbf{D}, \{q_j\}_{j=1}^{i-1}, \{occ_j\}_{j=1}^{i-1}, \mathcal{L}_{Setup}, \mathcal{L}_{Load}, \{\mathcal{L}_{Query,j}\}_{j=1}^{i-1},$
                      $\mathcal{T}_{Setup}, \mathcal{T}_{Load}, \{\mathcal{T}_{Query,j}\}_{j=1}^{i-1})$
    $res_i \leftarrow \mathcal{P}.\textsc{Query}(q_i, occ_i)$
$transcript \leftarrow \{\{res_i\}_{i=0}^{d}, \{\mathcal{T}_{Query,i}\}_{i=1}^{d}, \mathcal{T}_{Setup}, \mathcal{T}_{Load}, \mathtt{st}_{\mathcal{A}}\}$

Experiment $transcript \leftarrow \mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}(\lambda)$:
$(\mathbf{D}, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_0(1^\lambda)$
$(\mathcal{I}^{\mathcal{S}}, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_0(\mathcal{L}_{Setup}, \mathcal{L}_{Load})$
$res_0 \leftarrow \mathcal{P}.\textsc{Load}(\mathcal{I}^{\mathcal{S}})$
$\forall i \in \{1, \dots, d\}$:
    $(q_i, occ_i, \mathtt{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i(\mathtt{st}_{\mathcal{A}}, \mathbf{D}, \{q_j\}_{j=1}^{i-1}, \{occ_j\}_{j=1}^{i-1}, \mathcal{L}_{Setup}, \mathcal{L}_{Load}, \{\mathcal{L}_{Query,j}\}_{j=1}^{i-1},$
                      $\mathcal{T}_{Setup}, \mathcal{T}_{Load}, \{\mathcal{T}_{Query,j}\}_{j=1}^{i-1})$
    $(q_i^{\mathcal{S}}, occ_i^{\mathcal{S}}, \mathtt{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_i(\mathtt{st}_{\mathcal{S}}, \mathcal{L}_{Setup}, \mathcal{L}_{Load}, \{\mathcal{L}_{Query,j}\}_{j=1}^{i})$
    $res_i \leftarrow \mathcal{P}.\textsc{Query}(q_i^{\mathcal{S}}, occ_i^{\mathcal{S}})$
    if $res_i \neq \mathtt{abort}$: $res_i \leftarrow R_{q_i}$
$transcript \leftarrow \{\{res_i\}_{i=0}^{d}, \{\mathcal{T}_{Query,i}\}_{i=1}^{d}, \mathcal{T}_{Setup}, \mathcal{T}_{Load}, \mathtt{st}_{\mathcal{A}}\}$
```

Figure 4.3: *Security experiments for PPSS protocol $\mathcal{P}$*

As in the DORAM security analysis, we define the traces of the three procedures of ObSQRE (i.e., SETUP, LOAD and QUERY) as the information observed by the adversary throughout the execution of these procedures. For the ones running inside the enclave, namely LOAD and QUERY, the traces $\mathcal{T}_{Load}$ and $\mathcal{T}_{Query}$ are defined as in Def. 4.1, with the DORAM client being trivially replaced by the corresponding procedure; instead, since the SETUP procedure is executed at data-owner's side, its trace $\mathcal{T}_{Setup}$ is limited to the encrypted full-text index sent to the untrusted server, which is denoted as $\mathcal{I}$. We also define the leakage $\mathcal{L}$ as the information inferred by the adversary about the document collection $\mathbf{D}$, the substring queried q and the occurrences of q in $\mathbf{D}$; we split $\mathcal{L}$ in three components $\mathcal{L}_{Setup}$, $\mathcal{L}_{Load}$ and $\mathcal{L}_{Query}$, denoting the leakage inferred by the adversary from the traces $\mathcal{T}_{Setup}$, $\mathcal{T}_{Load}$ and $\mathcal{T}_{Query}$, respectively.

In our security definition, we consider a modified QUERY procedure that, instead of retrieving all the o_q occurrences of a string q in $\mathbf{D}$, allows to specify the number occ of occurrences to be retrieved. This procedure can be implemented by fetching occ entries instead of o_q ones from the SA Suf in the second phase of backwards search algorithm (lines 6-7 of Alg. 1.2). This modification allows to prove that ObSQRE is secure against an adversary that can choose the query to be performed after observing the traces of previous queries. This is a strong security notion, known as *chosen adaptive security*, as it guarantees that the protocol remains secure even if the adversary can somehow control ObSQRE operations (e.g., forcing to always querying the same string).

Definition 4.3 (PPSS Security Against Malicious Adversary)**.** *Given a security parameter λ, an integer $d \geq 1$, and a PPSS protocol $\mathcal{P}$ with trace $\mathcal{T} = \{\mathcal{T}_{Setup}, \mathcal{T}_{Load}, \mathcal{T}_{Query,1}, \dots, \mathcal{T}_{Query,d}\}$ and leakage $\mathcal{L} = \{\mathcal{L}_{Init}, \mathcal{L}_{Load}, \mathcal{L}_{Query,1}, \dots, \mathcal{L}_{Query,d}\}$, consider the two interactive experiments $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ and $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$, outlined in Fig. 4.3, between a challenger and an adversary $\mathcal{A}$ consisting of $d{+}1$ probabilistic polynomial time algorithms,* *i.e.,* $\mathcal{A} = \{\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_d\}$*. Throughout the experiments, the challenger may invoke the protocol* $\mathcal{P}$ *and a probabilistic polynomial time simulator* $\mathcal{S}$ *consisting of* $d+1$ *probabilistic polynomial time algorithms, i.e.,* $\mathcal{S} = \{\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_d\}$*; the adversary* $\mathcal{A}$ *can tamper with data and computation of the PPSS protocol as described in our threat model. Denoting as* $\mathcal{D}$ *a probabilistic polynomial time algorithm that, given the transcript* o *of an experiment, determines if* o *refers to* $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ *(*$\mathcal{D}(o) = 0$*) or* $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$ *(*$\mathcal{D}(o) = 1$*) experiment, the PPSS protocol* $\mathcal{P}$*, with leakage* $\mathcal{L}$ *and trace* $\mathcal{T}$*, is secure against malicious probabilistic polynomial time adversaries* $\mathcal{A}$ *if, for every possible* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that for every* $\mathcal{D}$*:*

$$\mathtt{Pr}(\mathcal{D}(o) = 1 | o \leftarrow \mathbf{Real}_{\mathcal{P},\mathcal{A}}) - \mathtt{Pr}(\mathcal{D}(o) = 1 | o \leftarrow \mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}) \leq \epsilon(\lambda)$$

where $\epsilon(\cdot)$ *is a negligible function.*

As for the security definition employed for our DORAMs, the indistinguishability between the transcripts of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ and $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$ experiments reported in Fig. 4.3 is sufficient to ensure the following security guarantees:

- The PPSS protocol $\mathcal{P}$ leaks to any malicious adversary no more information than the leakage $\mathcal{L}$
- The PPSS protocol can detect any data or computation tampering that affect the result of the protocol, in turn ensuring the correctness of the protocol against any misbehavior from a malicious adversary

ObSQRE provides the security guarantees of Def. 4.3 while exhibiting the information leakage $\mathcal{L}$ reported in the following statement:

Theorem 4.2. *For a document collection* $\mathbf{D}=\{D_1, \ldots, D_z\}$ *with* $z \geq 1$ *documents and* $d \geq 1$ *strings* $q_1, \ldots, q_d$*, assuming that a DORAM with the privacy guarantees outlined in Thm. 4.1 is employed, ObSQRE is secure according to Def. 4.3 with a leakage* $\mathcal{L} = \{\mathcal{L}_{Setup}, \mathcal{L}_{Load}, \mathcal{L}_{Query,1,\ldots,\mathcal{L}_{Query,d}}\}$ *defined as:*

- $\mathcal{L}_{Setup} = \{n = \sum_{i=j}^{z} \textsc{Len}(D_j)\}$
- $\mathcal{L}_{Load} = \{n\}$
- $\mathcal{L}_{Query,i} = \{m_i, occ_i\}, i \in \{1, \ldots, d\}$*, where* $m_i = \textsc{Len}(q_i)$ *and* occ_i *is the number of occurrences of* q_i *in* $\mathbf{D}$ *chosen to be retrieved by the remote user*

We prove this theorem in Appendix 3.2. Informally, the security guarantees of ObSQRE are based on the confidentiality and integrity guarantees given by Intel SGX and by the obliviousness of the substring search algorithm run inside the enclave. Indeed, the control flow of our oblivious backward search algorithm depends either on publicly known parameters (e.g, P or $|\Sigma|$) or from data found in the leakage $\mathcal{L}$; the accesses to all the data structures employed in our substring search algorithms are all oblivious, as they are performed either with a linear sweep over the data structure of through a DORAM. The information leakage $\mathcal{L}$ is inferred by the adversary from the size of the outsourced full-text index, which is proportional to the size n of the document collection, and from the number of iterations of the two loops of the backward search algorithm (Alg. 1.2), which leak the the length m_i of the substring searched q_i and the number occ_i of occurrences of q_i in $\mathbf{D}$ retrieved by the remote user, respectively. Finally, the correctness

Table 4.3: *Parameters chosen for our DORAMs. S is the stash size, A the eviction period, Z (resp. D) the max. (resp. min.) number of real (resp. dummy) blocks per bucket*

DORAM	Z	S	D	A
Path [142]	4	32	0	3
	8	41	0	8
Circuit [156]	3	8	0	1
Ring [128]	4	32	6	3
	8	41	13	8

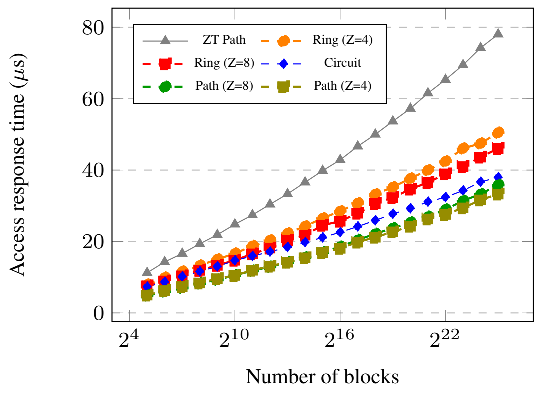
of the result against any adversarial misbehavior is ensured by the security guarantees of SGX and by the integrity-protection mechanism of our DORAMs: indeed, the latter ensures the authenticity of the data fetched from the DORAM tree, while all the other operations are performed inside the enclave, where the adversary cannot tamper with any data or code thanks to the security guarantees of Intel SGX technology.

4.4 Experimental Evaluation

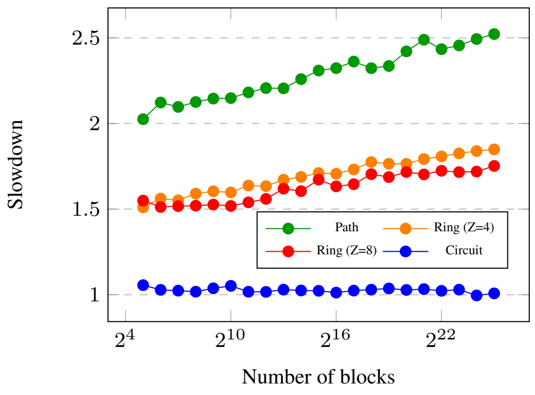
We realized a publicly available C++ implementation [133] of ObSQRE employing the Intel SGX SDK 2.5 [41]. To encrypt blocks in the DORAM, we employed the AES implementation of WolfSSL [143]. We performed all our tests on an Ubuntu 16.04 LTS server equipped with 64 GiB of RAM memory and an Intel Xeon E3-1220 v6 CPU clocked at 3 GHz, where SGX is available. To evaluate our substring search algorithms with different alphabets, we considered three datasets: the 21^{st} human chromosome [17] (`Chr` in short), which encodes DNA sequences employing 7 symbols of the FASTA format [38]; the `SwissProt` database [153] (`Prot` in short), which contains $550k$ human proteins, encoded with 25 symbols as a sequence of amino-acids; the Enron dataset [87] (`Enron` in short), which contains real emails of a financial firm over an alphabet of 96 ASCII characters.

DORAM Benchmarking

We started our experimental campaign by comparing the response time of the ACCESS procedure for our three DORAMs, excluding accesses to the position map in order to make these tests meaningful also for ODS. We instantiated each of these DORAMs with parameters chosen according to the configurations provided by the authors of the corresponding ORAM, except for Path DORAM where we employed the same configurations of Ring DORAM. For Path and Ring DORAMs, we considered two possible configurations to explore the trade-off between the size of a bucket and the eviction period A: indeed, while smaller buckets reduce the computational cost of DORAM procedures, evictions, which are the most expensive operations, are performed more frequently. The configurations employed for our tests are reported in Tab. 4.3; we empirically verified that no stash overflow occurs after 2^{30} round robin accesses with the chosen parameters. For all the configurations, we measured the response time to access one block, averaged over 1024 accesses, for DORAMs with 2^i, $i \in \{5, \ldots, 25\}$, real blocks storing 8 bytes of data each. In all the tests, we fully initialized all the blocks in the DORAM before measuring the response time.



(a) *Response time of accesses for our DORAMs and the Path DORAM of ZeroTrace [135]*



(b) *Ratio between response times of DORAMs and SORAMs accesses*

Figure 4.4: *DORAM Benchmark*

Figure 4.4(a) reports the results of our benchmark, showing that Path DORAM is the fastest one among our DORAMs. This outcome is motivated by the simplicity of the client operations w.r.t. Ring DORAM and by the higher eviction period than Circuit DORAM. Nonetheless, we observe that the response time of Circuit DORAM grows slower w.r.t. Path and Ring DORAMs, confirming that Circuit DORAM is asymptotically faster. It is worth noting the different impact of the eviction period and bucket size in Ring and Path DORAMs: indeed, in the former, evicting less frequently achieves better performance, even if each eviction is slower due to the larger buckets; conversely, since in Path DORAM both EVICTION and FINDBLOCK procedures become slower with larger buckets, a lower eviction period, which allows to employ smaller buckets, yields better performance. To compare our DORAMs with existing ones, we report in Fig. 4.4(a) the access response time of our own implementation of the Path DORAM proposed in ZeroTrace [135] (and employed with minor modifications in all the existing works [4, 111]), referred to as ZT Path DORAM. The comparison shows that all our DORAMs are faster than ZT one. In particular, our Path DORAM is about $2\times$ faster than ZT one, clearly showing the performance gain given by amortizing the cost of evictions over $A\geq 1$ accesses.

Willing to assess the overhead introduced by our oblivious clients in DORAMs, in Fig. 4.4(b) we compare their access response time with the one of Singly Oblivious RAMs (SORAMs), which correspond to the original versions of the ORAMs. For Path ORAM, we compare the configuration achieving best performance for the DORAM with a configuration for the SORAM suggested by authors in [142], i.e., $Z{=}4$ and $S{=}64$. The results in Fig. 4.4(b) show that the computational overhead introduced by oblivious clients is acceptable, being at most $2.5\times$. As expected, the slowdown of Circuit DORAM is negligible, given the simple modifications required to make the client oblivious. Conversely, Path DORAM exhibits the highest slowdown: this overhead mostly comes from making the EVICTION procedure oblivious and from the additional path that is written back to the DORAM tree during the FINDBLOCK one.

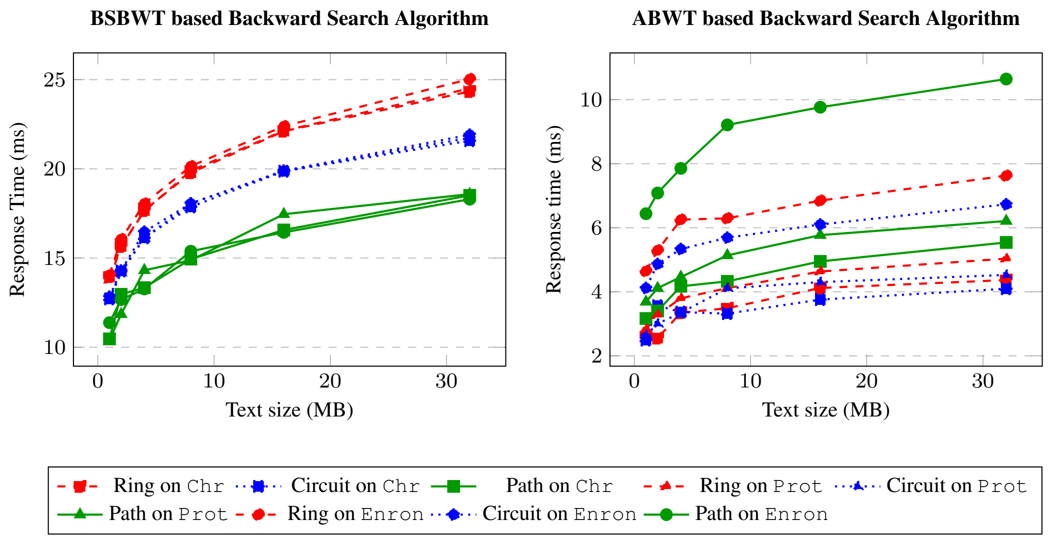


Figure 4.5: *Comparison of ObSQRE oblivious substring search algorithms for* `Chr`, `Prot` *and* `Enron` *datasets.*

ObSQRE PPSS Protocol Evaluation

We now show the results of our evaluation of the two proposed oblivious substring search algorithms, combining each of them with each of our DORAMs, and choosing, for Path and Ring ones, the most efficient configuration identified in our benchmark (Fig. 4.4(a)). We evaluated the response time to compute the number of occurrences of a substring with 24 characters for increasing sizes of the datasets, without considering the retrieval of the positions of the said occurrences as they depend neither on the specific substring search algorithm nor on the alphabet size. To achieve the maximum performance for the ABWT based algorithm, we performed an exhaustive parameter space exploration to find the optimal values for the sample period P and the factor C, which are employed to construct the ABWT and the recursive position map for the DORAM, respectively. Specifically, we identified, for each DORAM with $M{=}2^i$ real blocks, $i \in \{9,\ldots,25\}$, the value C minimizing the response time to recursively access the position map of the DORAM at hand. Then, for each sample period P, we stored the corresponding ABWT index inside a DORAM with $M{=}2^{\lceil \log(\lceil \frac{n+1}{P} \rceil) \rceil}$ blocks, building the position map with the optimal recursive factor C for a DORAM with M blocks, and we chose the sample period P minimizing the query response time.

The results of the evaluation for both our oblivious substring search algorithms are reported in Fig. 4.5. We observe that, regardless of the DORAM being employed, the ABWT based algorithm is by far the fastest, as its response time is about 3 to 5 times smaller than BSBWT one. This performance gap shows that it is more beneficial to access few DORAMs rather than employing a small block size; indeed, the performance gain of the ABWT based RANK procedure is due to the $\log_C(\lceil \frac{n+1}{P} \rceil)$ DORAMs accessed instead of the $\log_2(n)$ ones in the BSBWT based method. The comparison among different datasets reveals that the ABWT algorithm is more affected by the alphabet size $|\Sigma|$, as, regardless of the DORAM, the queries for the `Enron` dataset are slower than `Chr` and `Prot` ones. This is expected, as the size of the entries in the ABWT, and thus the block size of the DORAM storing it, depends linearly on $|\Sigma|$;

Table 4.4: *Performance of ObSQRE and less secure alternatives applied to the genomic and* `Enron` *datasets with different lengths of the queried substring (m). Response time to compute the number of occurrences is denoted as* **S***, while* **R** *denotes the time to retrieve all the corresponding positions*

Dataset	m	**#Occ.**	**ObSQRE (ms)**	**SGX+ SORAM (ms)**	**no SGX+ no ORAM (μs)**
Genome	**3050**	1	**S:** 1019 **R:** 0.6	**S:** 347 **R:** 0.3	**S:** 167 **R:** 0.2
Enron	**5**	2657	**S:** 1.6 **R:** 2.1	**S:** 0.6 **R:** 0.8	**S:** 0.8 **R:** 10.4
Enron	**13**	290	**S:** 5.0 **R:** 0.5	**S:** 1.7 **R:** 0.2	**S:** 3.1 **R:** 1.1
Enron	**20**	154	**S:** 7.8 **R:** 0.6	**S:** 2.8 **R:** 0.2	**S:** 5.1 **R:** 0.6

conversely, BSBWT algorithm is negligibly affected by $|\Sigma|$.

Concerning the different DORAMs employed to store the full-text index, the ABWT based algorithm achieves the best performance when combined with Circuit DORAM (blue lines in the right pane in Fig. 4.5), while Path DORAM outperforms the other ones in the BSBWT algorithm (green lines in the left pane in Fig. 4.5). However, while Path DORAM is the most efficient in our benchmarks in Fig. 4.4(a), it exhibits the largest slowdown when employed for the ABWT algorithm. This outcome is due to the low value of the factor C (comparatively with the values derived for the other DORAMs) which is identified as optimal for Path DORAM in the previously mentioned exhaustive parameter space exploration for the ABWT algorithm. Indeed, a low C implies a high number of deployed Path DORAMs to recursively store and access the position map. Since the only performance benefit given by a low value for C is that the blocks of all these DORAMs are smaller, our conjecture is that Path DORAM is more affected by the block size than other DORAMs; to validate our hypothesis, we evaluated the response time of our DORAMs with blocks of increasing size, hereby observing a much worse performance degradation in Path DORAM than in Circuit and Ring ones.

Once determined that ObSQRE achieves the best performance when the ABWT based backward search is paired with Circuit DORAM, we validated the practicality of this solution on two realistic use cases: the look-up of the occurrences of a DNA sequence corresponding to a protein in the entire human genome, whose size is approximately 3 GB, and the look-up of all the occurrences of three typical strings (i.e., *Fitch*, *Business Trip* and *Investment Portfolio*) in the financial domain over the whole `Enron` email corpus, whose size is about 1 GB. Furthermore, we evaluated the overhead of ObSQRE w.r.t. baseline solutions with weaker security guarantees: an application running the ABWT based substring search algorithm outside the enclave, which has no security guarantees; an application running the algorithm inside the enclave but employing Path SORAM (i.e., the fastest among our SORAMs) instead of Circuit DORAM, which is secure only if the critical leakage of memory access patterns inside SGX is ignored (as in SGX threat model). Indeed, even neglecting the information leakage coming from these side-channels, the adoption of a SORAM is needed to protect accesses to the full-text index and the suffix array, as these data structures must be necessarily stored outside the enclave because of the limited memory area that is reserved by SGX to enclave applications, which amounts to only about 100 MB. Table 4.4 outlines the results of this evaluation. Although the overhead incurred by ObSQRE over a solution with no security guarantees in both the use cases amounts to about 3 order of magnitudes, ObSQRE is only $3\times$ slower than a solution that ensures confidentiality of the data in the SGX threat model, which does not encompass information learned from side channels. Given the high security guarantees offered by ObSQRE, we deem this security cost as acceptable for real-world applications. Indeed, the results show the practicality of ObSQRE in real-world scenarios, as the occurrences of a protein over the whole genome are found in only 1.019 seconds, while the occurrences of short strings in the whole `Enron` corpus are found in just few milliseconds.

Part II

Digital Certificates

CHAPTER 5

Definition and Preliminaries

We start the second part of this manuscript by providing the definitions and the background information about digital certificates and the two formats considered in this work, namely the X.509 and the OpenPGP ones. We organize this chapter as follows:

- In Section 5.1, we recall the language theoretical notions that are employed in our analysis of the X.509 and OpenPGP formats
- In Section 5.2, we describe the format of X.509 digital certificates
- In Section 5.3, we briefly introduce the OpenPGP protocol and we describe in detail the format of its digital certificates and messages

5.1 Language Theoretical Concepts

Given a finite set of symbols $\mathbf{\Sigma}$, known as *alphabet*, a language $\mathbf{L}$ is defined as a set of elements, named *strings*, obtained by concatenating zero or more symbols of $\mathbf{\Sigma}$. Conventionally, the *empty string* is denoted as ε, while a portion of a string is known as a *factor*.

Given a language it is also possible to describe it via a generative formalism, i.e., a *grammar*. A grammar is a quadruple $G : (\mathbf{V}_n, \mathbf{\Sigma}, \mathbf{P}, S)$, with $\mathbf{V}_n$ a finite set of symbols named *nonterminals*, $\mathbf{\Sigma}$ the alphabet of the generated language, whose symbols are also named *terminals*, $\mathbf{P}$ a set of *productions* defined as pairs of strings obtained by concatenating elements of $\mathbf{V}{=}\mathbf{\Sigma}{\cup}\mathbf{V}_n$, and $S{\in}\mathbf{V}_n$ the *axiom* of the grammar. The productions are denoted as usual as $\alpha{\to}\beta$, where $\alpha \in \mathbf{V}_n^+$ is the Left Hand Side (LHS) of the production and $\beta \in \mathbf{V}^*$ is the Right Hand Side (RHS), with $\mathbf{V}^+$ representing the closure with respect to concatenation over $\mathbf{V}$, and $\mathbf{V}^*{=}\mathbf{V}^+{\cup}\{\varepsilon\}$. The generative framework of a grammar describes a string of the language as the results of a generation process starting from the string $w_0 = S$. At each generation step i, the process outputs a string $w_i \in \mathbf{V}^*$, $i \geq 1$. At the i-th step a production $p \in \mathbf{P}$ is randomly chosen among the ones having their LHS in w_{i-1}; then, the LHS of p in w_{i-1} is substituted with the RHS of p to derive w_i. The process stops when all the symbols in the sequence w_i are terminals (i.e., $w_i \in \Sigma^*$). Languages are classified according to the Chomsky hierarchy [37], depending on which constraints are holding on the productions of the grammar generating them. Language families generated from grammars with stricter constraints are included in all the families with weaker constraints. In the following, we describe the language families starting from the least constrained one.

Grammars with no restrictions on the both LHS and RHS of their productions are denoted as *unrestricted* grammars, and generate Context Sensitive with Erasure (CS-E) languages. Determining if a generic string over $\mathbf{\Sigma}$ belongs to a CS-E language is not decidable. Restricting the productions of a grammar so that the sequence of substitutions does not allow the erasure of symbols yields Context Sensitive (CS) grammars, which generate CS languages. Determining if a generic string over $\mathbf{\Sigma}$ belongs to a CS language is decidable, i.e., it is always possible to state whether a string over $\mathbf{\Sigma}$ can be generated by a given CS grammar. Restricting the productions of a grammar to have a single element of $\mathbf{V}_n$ on their LHS yields the so-called Context-Free (CF) grammars, which generate CF languages. Deterministic Context-Free (DCF) languages can be recognized by a DPDA and their generating grammars constitute a proper subset of non ambiguous CF grammars. Finally, restricting the productions of a grammar either to the ones of the form $\{A \rightarrow a, A \rightarrow aA\}$ or to the ones of the form $\{A \rightarrow a, A \rightarrow Aa\}$, where $A \in \mathbf{V}_n$, $a \in \mathbf{\Sigma} \cup \{\varepsilon\}$, yields *right-* or *left-linear* grammars, which generate Regular (REG) languages. We will denote such grammars as *linear* whenever the direction of the recursion in their productions is not fundamental.

Given a grammar G generating the language $\mathbf{L}$ and a generic terminal string $w \in \mathbf{\Sigma}^*$, the process of *parsing* consists both in determining if $w \in \mathbf{L}$ and in computing the sequence(s) of applications of the productions of G which generate(s) w. The sequence(s) of productions are depicted as a tree, called either *parse tree* or *abstract syntax tree*, with the axiom of the grammar being the root and the terminal symbols being the leaves. There exists a straightforward parsing algorithm of a CS language which is complete and correct, but its running time is exponential in the length of the input. As a consequence, a number of automated parser generation techniques with higher efficiency were defined for subsets of the CS grammar family. CF grammars are the mainstay of parser generation as it is possible to automatically generate a recognizer automaton for any language generated by them. In particular, given a generic DCF grammar it is possible to automatically generate a deterministic parser for the strings of the corresponding language [70]; such parser enjoys worst-case linear space and time requirements in the length of input. Linear grammars are optimal from a parsing standpoint, as it is possible to derive from them a parser that runs with constant memory and linear time requirements in the length of input. Moreover, a parser for a REG language can be generated starting from the definition of a *regular expression* [18].

In the remainder of this work, the terminal alphabet $\mathbf{\Sigma}$ will be the set of 256 values which can be taken by a byte (i.e., $\Sigma = \{0,1\}^8$), unless pointed out otherwise. Each terminal symbol will be denoted by two hexadecimal digits (e.g., the byte taking the decimal value 42 will be denoted as $\mathtt{2A}$). We will also employ the Extended Backus-

Naur Form (EBNF) to write the RHS of grammar productions.

5.2 Format of X.509 Digital Certificates

We now describe the X.509 format of digital certificates, whose specification is found in RFC5280 [39] and its complements [72, 122, 93, 152, 45]. Since the X.509 standard is described through the Abstract Syntax Notation 1 (ASN.1) meta-language, we introduce this notation before describing in detail the structure of an X.509 digital certificate.

ASN.1 Description

The ASN.1 meta-language allows to specify the structural constraints of a data format, describing it as a set of so-called Abstract Data Types (ADTs). ASN.1 is described in the set of ITU Recommendations (ITU-R) X.680-X.683 [79], while the encoding schemes for the ADTs are specified in ITU-R X.690–X.696 [79]. ASN.1 provides the means to express both the syntax of the ADT at hand, in a form akin to a grammar, and some semantic constraints concerning the values taken by an instance of such an ADT. To ease the comprehension of the analysis of X.509 from a language-theoretical perspective, we first provide a mapping between the syntactic-structure-specifying keywords of ASN.1 and the corresponding productions of an EBNF grammar. Subsequently, we highlight how the remaining, non purely syntactic features of ASN.1 act as constraints on the language of the instances of the described ADT in terms of semantics and describe the meta-language facilities exposed to ease the definition of a non ambiguous specification.

Syntactic Elements. An ASN.1 ADT can be regarded as a construct equivalent to a single EBNF grammar generating all the possible concrete data type instances as its language. The user-defined name of the ADT corresponds to the axiom of the EBNF grammar, while the productions are represented as structured data definitions with the ::= operator separating the left and right hand side, in lieu of the common $\rightarrow$.

The structure of an ADT may either be a single element, in case the type is *primitive*, or a composition of other types employing ASN.1 constructs in case it is *constructed.*

Primitive types in ASN.1 represent terminal rules of an EBNF grammar, i.e., rules where $\alpha{\rightarrow}\beta$, $\alpha{\in}\mathbf{V}_n^+, \beta{\in}\mathbf{\Sigma}^*$. The RHS of a primitive type definition is described by a single line ended by a specific keyword (e.g., `INTEGER`, `BOOLEAN`, `OBJECT IDENTIFIER`, `OCTET STRING`, `BIT STRING`), specifying completely its nature. An `OBJECT IDENTIFIER` is a sequence of decimal numbers separated by dots, of which a single decimal number is known as an *arc*.

Constructed types may either have a single user-defined name appearing on the RHS of their definition, in which case they act as the copy rules of an EBNF grammar (i.e., rules where $\alpha{\rightarrow}\beta, \alpha, \beta{\in}\mathbf{V}_n$), or an arbitrary ASN.1 syntactic construct may be used. The only exception to the aforementioned constructed type definition is the possibility of turning a primitive `OCTET STRING` or `BIT STRING` type into a constructed one through appending the keyword `CONTAINING` to it, followed by the description of its contents. Deriving a constructed type from an `OCTET STRING` forces its value in an ADT instance to be byte-aligned, and allows the designer to enforce the choice of the encoding rules to be employed for it with the `ENCODED BY` keywords followed

Table 5.1: *Equivalence between the notation of the ASN.1 RHS of a constructed type definition and the right hand side of an EBNF grammar production, with a set of sample ASN.1 types* `u, v, x` *matching the identically named strings $u, v, x \in \mathbf{V}^*$ in EBNF*

ASN.1 RHS	EBNF RHS
`SEQUENCE {u,v,x}`	uvx
`CHOICE {u,v,x}`	$u\|v\|x$
`u OPTIONAL`	$u\|\varepsilon$
`SET {u,v,x}`	$uvx\|uxv\|vux\|vxu\|xuv\|xvu$
`SEQUENCE OF u`	u^*
`SET OF u`	u^*
`SEQUENCE SIZE(1 .. MAX) OF u`	u^+
`SET SIZE(1 .. MAX) OF u`	u^+
`u(2 .. N)`	$u^2\|u^3\|\dots\|u^N$
`ANY`	Arbitrary definition
`ANY DEFINED BY u`	Arbitrary ASN.1 definition

by the encoding name (see ITU-R X.682) [79]. Table 5.1 shows a mapping between the ASN.1 structures appearing on the RHS of the data types that are found in X.509 specification and their matching EBNF notation, expressed with a set of sample ASN.1 types `u, v, x` and identically named strings $u, v, x \in \mathbf{V}^*$ in EBNF.

The ASN.1 notation specifies the common concatenation and union operators via the `SEQUENCE` and `CHOICE` keywords, respectively, while the syntactic constraint indicated by the `SET` keyword mandates that a set of ADTs may appear in any order, without repetitions. Post-fixing an ADT appearing in a RHS of a definition with the `OPTIONAL` keyword allows it to be either missing or present only once. Given an ASN.1 ADT `u`, the syntactic constraints imposed by the `SEQUENCE OF` and `SET OF` constructs, indicate a concatenation of zero or more instances of `u`, matching the star operator in EBNF. The ternary range operator in ASN.1, having the syntax `t(low ... high)`, where `t` is an ADT and `low,high` the range boundaries, is employed with two purposes: specifying the range of possible values of the instances of the primitive ADT to which they are appended, or indicating the concatenation of any number `low`$<n<$`high` of instances of the user-defined ADT preceding them. The bounds of a range operator are either constant values or the keywords `MIN` and `MAX`, which indicate that the minimum (resp., maximum) of the given range is interpreted as the smallest (resp., greatest) value that can be taken by the ADT on their left. ASN.1 allows to specify *size constraints* through the use of the keyword `SIZE`, followed by a range of allowed sizes. A common idiom in ASN.1 ADT declarations is to employ `MAX` as an upper bound for `SIZE`s to indicate that there is no upper bound on the size. As a consequence, the two common ASN.1 `SEQUENCE SIZE(1 .. MAX) OF` and `SET SIZE(1 .. MAX) OF` idioms in Tab. 5.1 represent a concatenation of one or more instances of the involved ADT `u`, corresponding to the EBNF cross construct. Finally, the ASN.1 keyword `ANY` allows to delegate the definition of the structure of a given ADT to another document, potentially not expressed in ASN.1. Specifying further the effect of the `ANY` construct with the `DEFINED BY` keywords, followed by the name of an ADT, enforces the fact that the specification should be expressed as an ASN.1 ADT.

Semantic Elements and Disambiguation Constructs. ASN.1 allows to describe semantic information on instances of an ADT either specifying a constant value for a given *primitive type* element, appending such value between round brackets on the line where the type appears, or specifying a so-called DEFAULT value. Appending the DEFAULT keyword allows to indicate that, in case an element is missing in an instance of the ADT, the recognizer should assume its presence nonetheless, and assign the semantic value present in the ASN.1 specification to it.

The expressive power of the ASN.1 allows the designer to specify an ADT corresponding to an *inherently ambiguous* language, i.e., a language for which no unambiguous grammar exists. An illustrative example of such a case is the following ADT t:

```
t ::= CHOICE {
s1 SEQUENCE{u(i..i),v(i..i),x(j..j)},
s2 SEQUENCE{u(j..j),v(i..i),x(i..i)}}
i ::= INTEGER(1 .. MAX)
j ::= INTEGER(1 .. MAX)
```

which has its instances belonging to the intrinsically ambiguous language $\mathbf{L} = \{u^i v^i x^j \vee u^j v^i x^i$ s.t. $u, v, x \in \mathbf{\Sigma},\ i{\geq}1, j{\geq}1\}$. To provide a convenient way to cope with ambiguities, ASN.1 introduces the so-called user-defined *tag* elements. A tag is a syntactic element, denoted as a decimal number enclosed in square brackets, which is prefixed to an ADT appearing in the RHS of a data description. Proper use of tags minimally alters the language of accepted ADT instances, while effectively curbing ambiguities. Sticking to the previous example, the inherent ambiguity of the sentences of the language of the ADT t can be eliminated adding two tags to its description:

```
t ::= CHOICE {
s1 SEQUENCE{ [0] u(i..i),v(i..i),x(j..j)},
s2 SEQUENCE{ [1] u(j..j),v(i..i),x(i..i)}}
i ::= INTEGER(1 .. MAX)
j ::= INTEGER(1 .. MAX)
```

Binary Encoding for ASN.1 ADTs

While ASN.1 allows to detail the structure of an ADT, it does not define how its instances should be encoded in a machine readable (i.e., binary) format. The encoding rules for an ASN.1 data type instance (see ITU-R X.690–X.696 [79]) define several formats portable across different architectures by mandating bit and byte value conventions and ordering of the encoded contents. Among these formats, X.509 certificates mostly employ the Distinguished Encoding Rules (DER) encoding, as a *constructed typed* field in the X.509 standard certificate ADT requires this encoding via the ENCODED BY keyword. DER encoded material may be further mapped to the fully printable base64 encoding, resulting in the so-called Privacy-enhanced Electronic Mail (PEM) format [95]. The DER encoding strategy represents an ASN.1 ADT instance as a stream of bytes which is logically split up into three fields: *identifier octets*, *length octets*, *content octets*.

The *identifier octets* field is employed to encode the ASN.1 tag value and whether the ADT instance at hand is a primitive or a constructed ASN.1 type. The tag value may be either the disambiguating user-defined one present in the ASN.1 ADT definition, or a so-called *universal tag* assigned by the DER standard to all ASN.1 primitive types and to the SEQUENCE, SEQUENCE OF, SET, SET OF, ANY constructed ones.

```
Certificate ::= SEQUENCE {
tbsCert            TBSCertificate,
signatureAlgorithm AlgorithmIdentifier,
signatureValue     BIT STRING }

TBSCertificate ::= SEQUENCE {
version [0] EXPLICIT Version DEFAULT v1,
serialNumber         CertSerialNumber,
signature            AlgorithmIdentifier,
issuer               Name,
validity             Validity,
subject              Name,
subjectPublicKeyInfo SubjectPubKeyInfo,
issuerUniqueID  [1] IMPLICIT UniqueId   OPTIONAL,
subjectUniqueID [2] IMPLICIT UniqueId   OPTIONAL,
extensions      [3] EXPLICIT Extensions OPTIONAL }

AlgorithmIdentifier ::= SEQUENCE {
algorithm  OBJECT IDENTIFIER,
parameters ANY DEFINED BY AlgorithmP OPTIONAL}

SubjectPublicKeyInfo  ::=  SEQUENCE  {
algorithm         AlgorithmIdentifier,
subjectPublicKey  BIT STRING }
```

Figure 5.1: *Portion of the description of the X.509 `Certificate` ADT and its fields*

The universal tag allows to identify the specific type of the DER encoded ADT. If a user-defined tag is present, its encoding is stored in the *identifier octets* field, while the encoding of the tagged ADT is stored in the *content octets* field. To provide a more succinct encoding, ASN.1 allows to specify that a given user-defined tag is `IMPLICIT`, i.e., that it should replace the tag of the tagged ADT in the encoding in the *identifier octets* field. On the other hand, the `EXPLICIT` keyword states that a user-defined tag should be encoded according to the default behavior.

The *content octets* field contains the actual encoding of the ADT instance at hand, while the *length octets* one stores the size of the content field as a number of bytes. The number of bytes constituting the *length octets* field varies in the 1 to 126 range. The encoding conventions for the length value are stated in the ITU-R X.690 [78]. A short form and a long form for the encoding of a length value are possible. The short form mandates the encoding of the length field as a single octet in which the most significant bit is 0 and the remaining ones encode the size of the *content octet* field from 0 to 127 bytes. The long form consists of one initial octet followed by one or more subsequent octets containing the actual value of the *length octet* field (i.e., the number of bytes of the *content octets* field). In the initial octet, the most significant bit is 1, while the remaining ones encode the number of length field octets that follow as an integer varying from 1 to 126. Thus, the number of bytes for the *content octets* field is at most $2^{126 \cdot 8}$. The *length octet* field value equal to 128 encoded as a single byte is forbidden in DER, while in other standard encoding rules it is reserved to indicate that an indefinite number of bytes will follow.

Description of X.509 Certificate Structure

We can now describe the format of an X.509 digital certificate by hinging upon the ASN.1 meta-syntax.

Certificate ADT. Figure 5.1 reports a shortened version of the X.509 standard, defining the main `Certificate` ADT. In the figure, field names start with a lowercase letter, while user-defined ADT names start with capital letters. The `Certificate` ADT is a concatenation of three fields: the material To Be Signed typed as `TBSCertificate`, the identification data for the signature algorithm typed as `AlgorithmIdentifier`, and a `BIT STRING` field containing the actual signature value.

TBSCertificate ADT. Considering the contents of the `TBSCertificate` ADT, the first two fields contain a version number typed as `Version` (an `INTEGER` in the range $\{0, \ldots, 2\}$), and an integer value which must be unique among all the certificates signed by the same CA, which is typed as `CertSerialNumber`. The third field, typed as `AlgorithmIdentifier`, contains the information to uniquely identify the cryptographic primitive employed to sign the certificate.

The `AlgorithmIdentifier` ADT is a concatenation of two fields: an ADT `OBJECT IDENTIFIER` (OID) typed field `algorithm` and an optional field named `parameters`, which is typed as `ANY DEFINED BY AlgorithmP`. The OID value allows to uniquely label the signature algorithm, as well as binding the format of its parameters, which are described in in [122, 72, 93, 152, 45] for a set of standardized signature algorithms. The `issuer`, `validity` and `subject` fields contain information on the CA issuing the certificate and the subject to whom it has been issued, together with the validity time interval of the certificate. `issuer` and `subject` fields are typed as a `Name` ADT: a `SEQUENCE OF` of `SET OF` structures containing a concatenation of two fields typed as OID and `ANY`, respectively. Despite the quite baroque definition, the `Name` ADT is indeed employed to represent a list of names for both the issuer and the subject which are typically expressed as printable strings prefixed with a standardized OID value stating their meaning (e.g., organization, country). The most relevant among these names is the so-called Common Name (CN), which usually corresponds to the identifier of an entity in the application domain where the digital certificate is employed (e.g., in HTTPS the CN in the `Subject` field corresponds to the URL of the website the digital certificate is bound to). Because of its relevance for identification purposes, the CN is a sensitive field targeted by many impersonation attacks [13, 107, 81].

The `subjectPublicKeyInfo` field provides both the public key bounded to the subject identity, and information on the employed cryptographic primitive in the form of a `BIT STRING` and an `AlgorithmIdentifier` typed field, respectively. Following the `subjectPublicKeyInfo` field, the `TBSCertificate` ADT includes two deprecated extra optional fields, containing further information about the issuer and the subject. These fields are tagged with tags `[1]` and `[2]` respectively, preventing a possible parsing ambiguity arising from only one of them being present.

Extension ADT. The `extensions` field concludes the definition of the `TBSCertificate` ADT. Most of the information of modern certificates is contained in it, and its presence is mandatory in the current version (v3) of X.509 certificates. As reported in Fig. 5.2, the `Extensions` ADT is a sequence of one or more `Extension` typed fields. Each `Extension` ADT is composed of an OID typed field identifying it unambiguously, and a `critical` field typed as `BOOLEAN` indicating, if `True`, that the certificate validation should fail in case the application either does not recognize

```
Extensions ::= SEQUENCE SIZE(1..MAX) OF Extension

Extension ::= SEQUENCE {
extnID    OBJECT IDENTIFIER,
critical  BOOLEAN DEFAULT FALSE,
extnValue OCTET STRING CONTAINING ...  ENCODED BY der }
```

Figure 5.2: *X.509 ADT description reporting the definition of the last field of the* `TBSCertificate` *ADT*

or cannot process the information contained in the subsequent `extnValue` field. The `extnValue` field stores the actual payload of the extension, thus its content vary depending on the extension at hand (and indeed its structure depends on the `extnID` field). An example of sensitive information contained in the `extnValue` field is the so-called `KeyUsage`, i.e., information stating which is the legitimate purpose of the subject public key in the certificate at hand (e.g., signature validation or encryption).

5.3 Description of OpenPGP Protocol and Format

Before delving into the description of the OpenPGP format, we first introduce the main concepts and the architecture of the OpenPGP protocol, which are necessary to understand the meaning and the purpose of the fields found in OpenPGP digital certificates and messages.

OpenPGP Protocol

The basic idea of the OpenPGP protocol to authenticate the ownership of public keys found in digital certificates is building a network of users which can be represented as a directed graph, known as the Web of Trust (WoT); specifically, each user's public key is a node, while an edge between a source node and a destination one represents a signature computed by the user associated to the source node on the public key associated to the destination node. Each user is identified by its `User-ID`, which is usually a string composed of a name and an email address. Each OpenPGP certificate contains a self signed public key, known as the *primary public key*. The corresponding *primary private key* is the one employed by the user when vouching for the authenticity of another public key through digitally signing it, effectively acting as a CA. Other public keys may be present, and bound to a primary key: they are known as *public subkeys*. Such subkeys may be employed during the actual communication, that is, to encrypt or sign messages. To this extent, the OpenPGP protocol employs a hybrid message encryption scheme, where an ephemeral, per message, cryptographic session key is generated as the first step of a message encryption. This ephemeral key is used to encrypt the message employing a symmetric cipher, and is in turn encrypted with one of the receiver's public subkeys that can be used for such *key encapsulation* method. All the subkeys are bound to the corresponding primary public key by a signature computed using the primary private key. Moreover, every private subkey usually signs a binding of the primary public key to a subkey. There are also revocation signatures, which are mainly needed to state that a primary key or a subkey is no longer in use or to invalidate a past signature.

The OpenPGP certificates are available on a public network of mutually synchronized servers, known as *keyservers* [73]. These servers provide OpenPGP certificates via the HTTP Keyserver Protocol (HKP) [1], optionally using the TLS transport layer. It is worth noting that OpenPGP certificates are intended to be append-only, meaning that whenever a certificate is exported to a keyserver, only new parts may be added, while the old ones cannot be modified. Nevertheless, a keyserver is not expected to provide integrity checks. Thus, the most trivial but really effective attack from a malicious keyserver is omitting parts of the certificate: for example, a keyserver may remove a revocation signature from a certificate, thus turning a revoked key into a valid one. Finally, each OpenPGP client has its own local storage for certificates, known as the *keyring* of the owner. Coupled with the keyring, an additional storage, the *trustDB*, contains the "level of trust" of the owner towards the users bound to the certificates in her/his keyring when they act as a CA.

OpenPGP Format

The OpenPGP standard [29] defines the format of two valid OpenPGP objects: *certificate*s and *message*s; their format is defined as a sequence of the basic elements known as *OpenPGP packet*s, and differ only in the kind of packets allowed into each object and how they are combined.

OpenPGP Packet Structure. An OpenPGP packet is split into two parts: the packet header, which includes metadata needed to process the packet, and the packet body. The OpenPGP standard specifies that version 3 and version 4 packets have to be considered valid. The latter version supports a wider range of packet types and employs a different way of encoding the length fields in the packet header. The packet header consists of a single byte, named *packet tag*, followed by a variable length field, named *body length*, which encodes the number of octets composing the body of a packet. A ver. 3 *packet tag* specifies the *version* with 2 bits, the *body length field encoding* with 2 other bits, and then the *packet type* over the remaining 4 bits. In particular, the 2-bit encoding conventions for the *body length field encoding* are as follows [29]:

$\texttt{00}_{\texttt{bin}}$: specifies a *body length* field made of one (2^0) octet, which is employed for packet bodies with length up to $(2^8)^{2^0}-1 = 2^8-1 = 255$ bytes;

$\texttt{01}_{\texttt{bin}}$: specifies a *body length* field made of two (2^1) octets, which is employed for packets with a length ranging from 2^8 to $(2^8)^{2^1}-1 = 2^{16}-1$;

$\texttt{10}_{\texttt{bin}}$: specifies a *body length* field made of four (2^2) octets, which is employed for packets with a length ranging from 2^{16} to $(2^8)^{2^2}-1 = 2^{32}-1$;

$\texttt{11}_{\texttt{bin}}$: specifies an unbounded length of the packet body. In this case, no *body length* field is present in the packet header, and the recognition of the end of the packet is implementation dependent (e.g., checking the End-Of-File marker as the last octet in the OpenPGP object).

A ver. 4 *packet tag* specifies the *version* with 2 bits, and the *packet type* with the remaining 6 bits. The length of the packet body is specified by the numerical value encoded in the first octet of the *body length* field in the packet header. The values of the 1st octet range in:

$\texttt{00000000}_{\texttt{bin}}$–$\texttt{10111111}_{\texttt{bin}}$ specify a 1 octet *body length* field, with the length of the packet body being the unsigned decimal value in $\texttt{0}_{\texttt{dec}}$–$\texttt{191}_{\texttt{dec}}$ encoded by the 1st octet itself;

$\texttt{11000000}_{\texttt{bin}}$–$\texttt{11011111}_{\texttt{bin}}$ specify 2 octets for the *body length* field, while the length of the packet body is determined as the unsigned decimal value ranging in $192_{\texttt{dec}}$–$8383_{\texttt{dec}}$, which corresponds to the sum of $192_{\texttt{dec}}$ with the value encoded by concatenation of the five least significant bits of the 1st octet and the bits of the 2nd (indeed $8383_{\texttt{dec}} = 192 + 2^{13} - 1$);

$\texttt{11111111}_{\texttt{bin}}$–$\texttt{11111111}_{\texttt{bin}}$ specify 5 octets for the body length field, and the unsigned decimal value which is binary encoded in the remaining four octets ranges from $8384_{\texttt{dec}}$ to $2^{32}-1$;

$\texttt{11100000}_{\texttt{bin}}$–$\texttt{11111110}_{\texttt{bin}}$ encode an unsigned decimal number n in the range $\texttt{224}_{\texttt{dec}}$–$\texttt{254}_{\texttt{dec}}$. The quantity 2^{n-224} defines the length of the packet, which is thus at most $2^{254-224} = 2^{30}$ octets long. This encoding is employed when the length of the content to be encased in a packet is a-priori unknown and thus this content is split into packets having this type of length field until the length of the remaining portion of the content can be encased in the body of a packet having one of the previously listed length types.

The length encoding strategies of ver. 4 packets are somehow similar to the ver. 3 ones, with the main difference being the replacement of the unbounded length option with a less troublesome encoding. We remark that the packet body has a specific structure depending on the packet type encoded in its tag. These various types of packets and their structures are detailed, for the ones relevant to our analysis, throughout the description of the format of OpenPGP certificates and messages.

OpenPGP Certificate Format. In the following, we describe the format of OpenPGP certificates as in RFC 4880 [29], which is reported in Fig. 5.3(a) by employing the EBNF notation. In the EBNF grammar found in Fig. 5.3(a), the terminal symbols represent specific OpenPGP packets.

A minimal certificate is specified by the sequence of at least two terminal symbols: `PrimaryKey` and `UserID`. The former one is encoded as a `Public-Key Pkt` packet, including the primary public key and the corresponding cryptographic parameters. The latter is encoded as a `User-ID Pkt`, including user identity data. Between these two packets, there may be a set of signature packets. In the OpenPGP format, each signature is stored in a `Signature Pkt` packet, which has a specific field (i.e., `Signature Type`) that determines the class of signature the packet refers to (e.g., a revocation signature or a signature binding a subkey to the primary one). The signatures that may be found between `Primary Key` and the `UserID` packets are a list of `RevocationSelfSignatures`, each of which issued by the same private key authenticating the revoked binding, and a list of `DirectKeySignatures`, which are used to authenticate some information, stored as auxiliary data in the packet of the signature, about a key, without binding them to a specific `UserID`. For instance, a `DirectKeySignature` may be employed to authorize a set of keys other than the signed ones to issue revocation signatures on the key the signature refers to. Following the `UserID`, there is a list of optional `Signatures` which acknowledge the

```
PGPCertificate → PrimaryKey(RevocationSelfSignature)* (DirectKeySignature)*
(UserID(Signature)*)+ (UserAttribute(Signature)*)*
(Subkey(BindingSignatureRevocation)* PrimaryKeyBindingSignature)*
```

(a)

```
PGPMessage → Plaintext | Encrypted | Signed
Encrypted → SessionKeyCtx EncryptedMessage
SessionKeyCtx → SymmetricallyEncryptedCtx | AsymmetricallyEncryptedCtx
EncryptedMessage → Ciphertext | IntegrityProtectedCiphertext MessageMDC
Signed → Signature PGPMessage | OnePassSignature PGPMessage Signature
```

(b)

Figure 5.3: *EBNF specification of the OpenPGP Certificate format (a); and the OpenPGP Message format (b). Non-terminal symbols are reported with italic typeface, while the terminal ones, which corresponds to OpenPGP packets, are denoted with roman typeface. Mandatory symbols in the OpenPGP objects are reported with a bold typeface*

binding between the primary public key and its owner. We remark that the OpenPGP format provides four different types of signature to authenticate such binding, each specifying a different amount of efforts by the signer to actually verify the authenticity of such binding. Additional information about a user identity (e.g., a picture) may be specified in `UserAttribute`, encoded as a `User-Attribute Pkt`, and cryptographically bounded to the primary key by optional subsequent `Signatures`. Finally, an OpenPGP certificate terminates with a list of zero or more public `Subkeys` followed by an optional sequence of `BindingSignatureRevocations` and by a `PrimaryKeyBindingSignature`. The latter kind of signatures cryptographically binds the public subkeys to which they refer with the primary public key, while the former kind revokes such binding. A `Subkey` can be revoked only with a signature from its primary key, unless additional keys are authorized to issue revocation signatures in the corresponding `PrimaryKeyBindingSignature` packet. A `Subkey` is encoded as `Public-Subkey Pkt`, which has the same body structure of `Public-Key Pkt`.

The `Signature Pkt` body definitely exhibits the most complex structure among OpenPGP ones, which is now described in detail due to its relevance in our analysis of the OpenPGP format. Its body contains four 1-octet fields (i.e., including version number, signature class, public key cipher and hash algorithm, respectively) followed by a pair $\langle S_1, S_2\rangle$ of sequences of *sub-packets*. Sub-packets are intended to provide additional data decorating the signature itself and, similarly to packets, are specified by a header and a body. The header is partitioned in a *sub-packet body length* field and in a 1-octet *sub-packet type* field. *Sub-packet body length* field can be 1, 2 or 5 octets long, following the same encoding specified for the *body length* fields of ver. 4 packets. The content of a `Signature Pkt` up to S_1 (included) is fed to the signature algorithm, whilst sub-packets in S_2 are not meant to be part of the signature. Both S_1 and S_2 are prefixed with a 2-octet field specifying their size. It is worth noting that any type of sub-packet may be included in both sequences, except for the `Signature Creation Time Subpkt`, which must be placed in S_1. Finally, more than one sub-packet of the same type may be in the same sequence or `Signature Pkt`.

OpenPGP Message Format. The OpenPGP message format [29] is summarized in Fig. 5.3(b), where the terminal symbols denote a specific type of OpenPGP packet.

An `OpenPGPMessage` non-terminal symbol can be replaced with a `Plaintext` terminal symbol, an *`Encrypted`* non-terminal symbol, or a *`Signed`* non-terminal symbol. In the former case, a `Literal Pkt` encodes the original message prefixed with a field denoting its type (binary or textual). An *`Encrypted`* non-terminal symbol is defined as a *`SessionKeyCtx`* followed by an *`EncryptedMessage`*, to capture the sequence of data made of an encrypted payload preceded by the value of session key employed to obtain it. The session key is encrypted either with another symmetric key generated from a passphrase (`SymmetricallyEncryptedCtx`), or with one of the receiver's public subkeys (`AsymmetricallyEncryptedCtx`). An *`EncryptedMessage`* can be either the `Ciphertext` terminal symbol or the pair of terminal symbols: `IntegrityProtectedCiphertext MessageMDC`. In the former case, an encrypted OpenPGP packet containing the plaintext message (e.g., a `Literal Pkt`) is found in the body of the `Ciphertext` OpenPGP packet, while in the latter such a copy is also followed by a *modification detection code* (MDC), which is generally a hash, to provide message integrity.

A *`Signed`* message has two possible structures. The simplest one prefixes the `Signature` terminal symbol to an entire *`PGPMessage`*. The other one, employed to achieve a *single pass signature verification*, prefixes a `OnePassSignature` terminal symbol (encoded as a `One-Pass Signature Pkt`) and the *`PGPMessage`* to the `Signature` terminal symbol. The `One-Pass Signature Pkt` contains the parameters of the hash and public key algorithms, to start the execution of the signature verification algorithm prior to receive the actual signature data. Note that *`PGPMessage`* can be recursively expanded in a *`Signed`* message, therefore multiple signatures can be put on the same `Plaintext` or *`EncryptedMessage`*.

CHAPTER 6

Novel Regular Format for X.509 Digital Certificates

In this chapter, we describe our proposal for a novel format for X.509 digital certificates. Our format is designed to ease the parsing of X.509 digital certificates and it is described by a REG grammar, which implies that a parser with optimal complexities and sound correctness guarantees can be automatically generated from the grammar specification [70]. In addition, we report the experimental validation of a parser automatically generated from a REG grammar with the `yacc` [145] toolchain. Before delving into the description of our format, we report the parsing issues of X.509 digital certificates identified in [13] that mostly drive the design of our format.

6.1 Parsing Hindrances in X.509 Format

In this section, we analyze the X.509 certificate structure from a language theoretic standpoint. In particular, we highlight the portions of the certificate which hinder and harden the design of a grammar amenable to automatic parsing generation algorithms. These issues mostly drive the design of our novel format, with the extent of removing them to obtain an efficient and simple parser for X.509 digital certificates. We remark that, although the set of DER encoded X.509 digital certificates is formally a REG language, since a certificate (and thus any ADT specifying a portion of it) cannot be longer than 2^{126*8} bytes, we cannot practically rely on this fact to build an efficient parser. Indeed, although any ADT with no ASN.1 specified constraints on its size may be theoretically recognized by enumerating all the possible binary strings with at most 2^{126*8} bytes, this strategy cannot be adopted in practice; therefore, when in the following we argue that a portion of X.509 certificate requires a recognizer more powerful than a regular parser, it is implied that recognizing the portion at hand by enumerating all the possible strings is infeasible.

***Length octets* in DER Encoding**. While parsing DER encoded ASN.1 ADTs, a recognizer must match the number of bytes encoded by a *length octets* field with the actual length of the *content octets* field. Since both constructed and primitive ADTs are encoded with *length octets*, then parsing a constructed ADT requires to simultaneously compute the number of *content octets* of the constructed ADT and the number of *content octets* of at least one other ADT found in the *content octets* of the constructed ADT at hand. For instance, consider the following DER encoded ADT:

```
Item ::= SEQUENCE {
        key    INTEGER,
        valid  BOOLEAN}
```

While parsing the `key` field in the `Item` ADT, a recognizer must simultaneously modify two counters for each byte of the *content octets* of `key`: one counter to keep track of the number of bytes left in the *content octets* of the `INTEGER` field, and a second counter to keep track of the number of bytes left in the content octets of the `Item` ADT. In the X.509 format, except for `Certificate` ADT, any primitive or constructed ADTs is "nested" in another constructed ADT, that is it is found in the *content octets* of another ADT; therefore, while parsing `Certificate` ADT, one counter for each nested ADT must be simultaneously handled. This is impossible with a DPDA, as to modify one of the counter we lose any information about the other one, in turn requiring a CS recognizer.

Issue 6.1 (Matching Multiple length octets)**.** *Since an X.509 digital certificate is a constructed ASN.1 ADT, then multiple ADTs are simultaneously processed during the parsing of the certificate at hand. Therefore, checking the correctness of the number of bytes found in the length octets of these ADTs requires to simultaneously handle multiple counters, in turn implying that a CS recognizer is needed.*

Matching Repetitions of Long Strings. The need for a CS recognizer is also introduced by the same kind of constraint in two different portions of the certificates. This constraint is the need to check repetitions of arbitrarily long strings. First, consider the `signatureAlgorithm` field in `Certificate` ADT and the `signature` field in `TBSCertificate` ADT, both typed as `AlgorithmIdentifier`. While the latter is found in the portion of the certificate which is signed, the former is not. Therefore, it is expected that these two fields have the same content; nonetheless, since an `AlgorithmIdentifier` ADT contains the `parameters` field which is typed as an `ANY` ADT, this field may contain arbitrarily long string of bytes, in turn requiring a CS recognizer to check the equality of two fields typed as `AlgorithmIdentifier`, such as `signatureAlgorithm` and `signature` ones.

Issue 6.2 (Matching `AlgorithmIdentifier` ADT)**.** *Checking the equality of the content of the* `signatureAlgorithm` *and* `signature` *fields, both typed as* `AlgorithmIdentifier`*, requires to compare arbitrarily long strings, hence requiring a CS recognizer*

Similarly, the presence of some portions of the certificate is mandatory if the certificate is self-issued, which means that the issuer and the subject are the same entity. Recall that both `issuer` and `subject` are typed as `Name` ADT, which is an arbitrarily long sequence of names, which are generally printable strings, referring to the issuer or subject entity, coupled with an OID identifying their meaning. Therefore:

Issue 6.3 (Validating self-issued certificates)**.** *Determining if a certificate is self-issued requires to match the content of* `issuer` *and* `subject` *fields, which are both arbitrarily long string of bytes, in turn requiring a CS recognizer*

Parsing Ambiguities and Inconsistencies. Due to looseness in standard specification of some fields, there are also parsing ambiguities in the format. First, consider the `signatureValue` field in the `Certificate` ADT. Despite the X.509 standard is typing this field as a primitive `BIT STRING`, some standardized signature algorithms require it to be a constructed field (e.g., the DSA and ECDSA cryptographic primitives [152, 45]), in turn giving way to an ambiguity in its interpretation. The same kind of issue arises in `SubjectPublicKeyInfo` field in `TBSCertificate` ADT. Indeed, the public key field is typed as a `BIT STRING`, instead of a constructed ADT. These ambiguities may lead to security issues when parameters for a given cryptographic primitive are either misinterpreted as valid for another one or simply parsed incorrectly [47, 3]. A similar issue is introduced by the `extnValue` field in `Extension` ADT, which is typed as `OCTET STRING` but contains the extension data, which is usually a constructed ADT. Nevertheless, DER forbids the encoding of `OCTET STRINGS` as constructed types (see ITU-R X.690) [78], in turn forcing an inconsistency in the way `OCTET STRINGS` containing the value of an `extnValue` field should be treated during parsing (due to the `CONTAINING` keyword in the X.509 specification). We note that a less problematic definition of the field would have involved a dedicated constructed ADT for the `extnValue` field, which should have had its structure specified according to the value of the `extnID` field.

Issue 6.4 (Constructed `BIT STRING` and `OCTET STRING` Types)**.** *The adoption of* `BIT STRING` *(resp.* `OCTET STRING`*) primitive ADT in the* `signatureValue` *and* `SubjectPublicKey` *(resp.* `extnValue`*) fields of the X.*509*, which may sometimes contain other ADTs, lead to a parsing ambiguity, as such* `BIT STRING`*s (resp.* `OCTET STRING`*s) can be recognized either as primitive or as constructed ADTs, depending on the parser at hand*

Unmanageable Number of Rules. Recall that the `Extensions` ADT is a sequence of `Extension` ADTs. Such a sequence of ADT has no constraint on their order, as they all share the same ADT. However, a constraint expressed in natural language in the X.509 standard mandates that 2 `Extension` typed field instances with the same `extnID` field cannot appear, in turn providing a concrete hindrance to its representation in grammar form. Indeed, an exponential number of productions would be required to generate unique `Extension` instances in any possible order. Considering that the number of X.509 standardized extensions is 17 [77], the required grammar would have at least 2^{17} productions, which is hardly manageable by a designer.

Issue 6.5 (Extension Uniqueness)**.** *Since no extensions in a digital certificate can share the same* `extnID`*, the grammar generating all the subsets of the* 17 *X.*509 *standard extensions [77] appearing in any possible order requires at least* 2^{17} *productions*

6.2 Description of Our X.509 Format

We provide the formal specification of our novel format for X.509 digital certificates in terms of a syntactic grammar $G = (V_n, V_t, P_s, \langle\text{CERTIFICATE}\rangle)$, and a lexical

$$
\begin{array}{l}
\mathbf{cert} \to \mathbf{preamble\,token\_list} \\
\mathbf{preamble} \to \texttt{0x30 0x07 0x30 0x05 0xA0 0x03 0x02 0x01 0x04} \\
\mathbf{token\_list} \to \mathbf{token\ token\_list}\ |\mathbf{token} \\
\mathbf{token} \to (\mathbf{ctr0}|\mathbf{ctr1}|\ldots|\mathbf{ctr126}\,)^{*}\ \mathbf{token\_body\ term} \\
\mathbf{ctr0} \to \texttt{0x80} \qquad\qquad \mathbf{term} \to \texttt{0xFF} \\
\mathbf{token\_body} \to \mathbf{integer}\,|\,\mathbf{rsa\_oid}\,|\,\mathbf{ascii\_string} \\
\mathbf{integer} \to \mathbf{inttype\,payload} \qquad\qquad \mathbf{inttype} \to \texttt{0x02} \\
\mathbf{payload} \to ([\texttt{0x00} - \texttt{0xFE}]|\texttt{0xFF 0xFF})^{+} \\
\mathbf{rsa\_oid} \to \texttt{0x05 0x2A 0x86 0x48 0x86 0xf7 0x0d 0x01 0x01 0x01} \\
\mathbf{ascii\_string} \to \texttt{0x06}\ [\texttt{0x20} - \texttt{0x7F}]^{+}
\end{array}
$$

Figure 6.1: *Lexical grammar of the proposed certificate format. The lexer yields to the parser the tokens contained in the token list. For some tokens, e.g., OIDs, only a representative sample is reported*

one $G_l = (V_t, \Sigma, P_l, \mathbf{cert})$, as it is usual for modern technical languages. We recall that we consider the set of digital certificates compliant to our format as a language defined over the alphabet $\Sigma = \{0,1\}^8$, that is the set of 256 possible values for a single byte. The lexical grammar describes how the input alphabet is transformed into structured symbols V_t, the *tokens*, which are subsequently used as terminals by the syntax grammar. We will indicate the elements of V_t as strings of boldface lowercase alphabetic characters, plus the underscore, while the nonterminals V_n of the syntax grammar are reported as angular-brackets encased, uppercase strings, e.g., $\langle\mathrm{A}\rangle$.

Format Lexicon

The lexical grammar of the proposed format, of which the most relevant portions are reported in Fig. 6.1, is designed to provide a strategy for a gradual adoption of the format, and solve the parsing issues highlighted in Sec. 6.1. From a lexical standpoint, a certificate is composed of a fixed-length preamble followed by a non empty list of tokens having the `0xFF` character as a list separator. The fixed length preamble contains the representation of the DER encoded prefix of an X.509 certificate up to the version field, included, with the version set to 4. As a consequence any existing library will recognize our format as a new version of X.509 and, if such a format is not yet supported, will point out the error gracefully. Such a strategy allows a gradual roll-out of the new certificates, which is currently considered a crucial factor for adoption [16], as certificates are re-generated by the CAs either upon expiration of the current ones or as a transition policy. The use of a unique termination character (i.e., `0xFF`) avoids employing length fields, removing the need for a context sensitive recognizer reported in Issue 6.1. The contents of each token begins with a single byte encoding its type, which is followed by the payload of the token at hand. Since the tokens are separated by the presence of the `0xFF` delimiter, the token encoding does not allow a single `0xFF` to be present in their payload; in case such a byte value needs to be represented, a pair of `0xFF` characters is used. Each token can be prefixed by one or more *control byte*s, fulfilling the role of ASN.1 tags, i.e., preventing syntactic recognition ambiguities at parsing stage between two tokens of the same type. To avoid the introduction of recognition ambiguities between control bytes and type encoding, the former have their first bit set, while the latter have the first bit clear. Such an encoding allows 127 possible control bytes, and 127 possible primitive types, which we found plentiful to describe the language at hand. We note that the use of control bytes allows us to remove the concept of constructed data types in our format. Indeed, at encoding level, the constructed types are redundant delimiters of the structures they represent, unless the end of such structure cannot be unambiguously inferred from the next token found; nonetheless, in the latter case we can employ control bytes to unambiguously mark the end of the structure at hand. We illustrate this concept through two examples.

1. Consider the following ASN.1 definition of the `Tuple` ADT:

   ```
   Tuple ::= SEQUENCE{
           Values  SEQUENCE OF INTEGER,
           IsValid BOOLEAN
           }
   ```

   In this case, the *length octets* of the `SEQUENCE` ADT are not necessary to identify the end of the `Values` field: indeed, this can be unambiguously inferred as soon as a `BOOLEAN` ADT is found instead of an `INTEGER` one. In this case, erasing the tag and *length octets* of the `SEQUENCE` ADT in the DER encoding of the `Values` field does not affect the proper recognition of this syntactic structure

2. Consider the following alternative definition of the `Tuple` ADT:

   ```
   Tuple ::= SEQUENCE{
           Values SEQUENCE OF BOOLEAN,
           IsValid BOOLEAN DEFAULT TRUE
           }
   ```

   In this case, the *length octets* of the `SEQUENCE` ADT are necessary to determine the end of the `Values` field, that is the *length octets* actually act as a delimiter of the `SEQUENCE` ADT. In this case, erasing the tag and *length octets* of the `SEQUENCE` ADT in the DER encoding of the `Values` field would lead to the impossibility of determining if the last parsed `BOOLEAN` ADT represents the `IsValid` field or the last element of `Values` field. Nonetheless, if we introduce an ASN.1 tag before the `IsValid` field, then we can safely erase the tag and *length octets* of the `SEQUENCE` ADT in the DER encoding of the `Values` field, as in the previous example

Therefore, in our own format, to simplify the binary encoding we decide to remove constructed ADTs from the lexical grammar, resorting to a proper usage of control bytes in the syntactical grammar to avoid introducing ambiguities in the format. We observe that the removal of constructed ADTs from our lexical grammar implicitly solves ambiguities mentioned in Issue 6.4, since interpretation of a payload as a composition of primitive types is no longer possible. Indeed, in the troublesome cases mentioned in Issue 6.4 where other ADTs are found in the payload of a primitive type, our syntax grammar will be defined to expect a sequence of tokens rather than a single one.

Statement 6.1 (Regular Lexicon)**.** *The proposed lexical grammar is REG. Indeed, all the productions of the grammar in Fig. 6.1 having only terminal symbols in their RHS generate a REG sub-language. As a consequence of this, and the fact that regular languages are closed with respect to union and concatenation, also the productions*

$$\begin{aligned}
&\langle\text{CERTIFICATE}\rangle \rightarrow \textbf{preamble}\,\langle\text{HASH}\rangle\,\langle\text{SERIAL}\rangle\,\langle\text{ISSUER}\rangle\,\langle\text{VALIDITY}\rangle\,\langle\text{SUBJECT}\rangle\\
&\qquad(\langle\text{PK\_CERTSIG}\rangle\,\langle\text{INTERM\_PORTION}\rangle \mid \langle\text{PK\_NOCERTSIG}\rangle\,\langle\text{LEAF\_PORTION}\rangle)\\
&\qquad\langle\text{SIGNATURE}\rangle\\
&\langle\text{HASH}\rangle \rightarrow \textbf{hash\_oid}\,\textbf{any} \qquad\qquad \langle\text{VALIDITY}\rangle \rightarrow \textbf{time}\,\textbf{time}\\
&\langle\text{ISSUER}\rangle \rightarrow \langle\text{NOREPEATABLE\_OID}\rangle\,(\textbf{oid}\,\textbf{any})^{*} \mid (\textbf{oid}\,\textbf{any})^{+}\\
&\langle\text{SUBJECT}\rangle \rightarrow \langle\text{ISSUER}\rangle\,\langle\text{SUBJ\_ALT\_NAME}\rangle? \mid \langle\text{CRITICAL\_SUBJ\_ALT\_NAME}\rangle\\
&\langle\text{INTERM\_PORTION}\rangle \rightarrow \langle\text{NOSELF\_SIG}\rangle\,\langle\text{NOSS\_INTERM\_EXTN}\rangle \mid\\
&\qquad\langle\text{SELF\_SIG}\rangle\,\langle\text{SS\_INTERM\_EXTN}\rangle\\
&\langle\text{LEAF\_PORTION}\rangle \rightarrow \langle\text{NOSELF\_SIG}\rangle\,\langle\text{NOSS\_LEAF\_EXTN}\rangle
\end{aligned}$$

Figure 6.2: *Portion of the syntactic grammar of a certificate describing the axiom production and the first level of recursion*

***token_list**, **token**, **token_body**, and **integer** generate sub-languages which are REG. Consequentially, for the same reason, the axiom of the lexical grammar **cert** generates a regular language.*

Format Syntax

The syntactic grammar for our proposed certificate format, of which the most significant structures are reported in Fig. 6.2, is designed to tackle the issues outlined in Sec. 6.1 and introduces some additional constraints aimed at preventing possible attacks. The $\langle\text{CERTIFICATE}\rangle$ nonterminal, which is the axiom of the grammar, starts by generating the fixed preamble of the certificate, followed by the hash algorithm employed to compute the signature, so that it is possible to perform the computation of the hash of the message while parsing the certificate contents. Such a fact allows a single-pass certificate validation, yielding good efficiency and limited memory requirements. All the contents of the certificate, save for the initial preamble, are to be hashed and signed by the CA. The certificate structure continues with the certificate serial number, the information concerning the issuer, the certificate validity and the subject of the certificate. We impose a uniqueness constraint on the distinguished names employed in issuer and subject (e.g., common name, organization name), due to impersonation attacks based on duplication of these names [81]. These ones are the set of names defined as mandatory to be supported in [39](page 21); to ease checking of the uniqueness constraint, we impose that these mandatory names should always appear in the same order. Indeed, for a set of n elements, ensuring that each of them appears at most once would require at least 2^n productions, in turn making the grammar design hardly manageable even for few elements; instead, if an order of appearance is established, then the uniqueness constraint can be checked with a single EBNF production. In our grammar, the non terminal $\langle\text{NOREPEATABLE\_OID}\rangle$ is employed to generate this set of standard defined distinguished names, ensuring also their uniqueness. Furthermore, we allow the presence of a set of arbitrary distinguished names after the standard defined ones.

Concerning the $\langle\text{SUBJECT}\rangle$ non terminal, we note that the subject distinguished names may be followed by the *s*ubject alternative name extension. Conversely, in X.509 v3 format all extensions are found at the end of the certificate, immediately preceding the signature on the digital certificate. In our format, we move the subject alternative name extension immediately after subject distinguished names because it simplifies the syntactical checking of an existing constraint in the X.509 standard, which mandates that this extension must be present and must be marked as critical in case there are no names in the subject field of the certificate. Since in X.509 v3 format several extensions exhibit constraints on their presence or on their content that are introduced by the content of previous fields in the certificate, by moving such extensions immediately after the field they depend upon we ease the syntactical checking of each of these constraints: indeed, we manage to avoid the simultaneous check of all of them, which may lead to a blowup in the design complexity of the grammar.

Following this design principle, we also place the *extended key usage* and *key usage* extensions between the public key algorithm OID and the public key parameters, as the content of these extensions is dependent on the public key algorithm being employed. In contrast with X.509 v3, we require the presence of the *key usage* extension to be mandatory, since it is crucial for security. Indeed, X.509 v3 states that if such an extension is missing, a certificate may be used as an intermediate one, which is prone to easy but dangerous misuses. The portion of the certificate containing the public key, which includes also the *extended key usage* and *key usage* extensions, is generated in the grammar by either $\langle \text{PK\_CERTSIGN} \rangle$ or $\langle \text{PK\_NOCERTSIGN} \rangle$ nonterminal symbol, depending on whether the public key can be employed to verify signatures on other certificates or not, an information stated in the *key usage* extension. The syntactic structure of the remaining portion of the certificate significantly differs among these two cases, as the content of several extensions found in this portion depends on whether the certificate can act as an intermediate one in a certificate chain or not (e.g., the `ca` flag in *basic constraints* extension must be set to true in the former case). A similar difference in the certificate structure is subsequently introduced to explicitly represent whether a certificate is self-signed or not, resulting in three nonterminals, $\langle \text{SS\_INTERM\_EXTN} \rangle$, $\langle \text{NOSS\_INTERM\_EXTN} \rangle$, and $\langle \text{NOSS\_LEAF\_EXTN} \rangle$, that generate the remaining portion of the certificate fulfilling the constraints imposed by the content of the previous fields. Note that the remaining combination of features (i.e., $\langle \text{SS\_LEAF\_EXTN} \rangle$) is discarded as a self-signed certificate which can be used only as a leaf of a certificate chain is pointless. We remark that the presence of an explicit field specifying if a certificate is self-signed or not allows to determine if a certificate is self-issued without the need for a CS recognizer (Issue 6.3).

Each of the three possible nonterminals generates the portion of the certificate containing the remaining extensions with a specific set of constraints. In any case, the remaining extensions are spit in two sequences:

$$\langle \text{EXTN} \rangle \to \langle \text{STD\_EXTN} \rangle \ \langle \text{CUSTOM\_EXTN} \rangle$$

The $\langle \text{STD\_EXTN} \rangle$ nonterminal generates any subset of the extensions described in the X.509 standard (except for the ones placed earlier in the certificate, such as *key usage*). We establish a fixed order of appearance for standard extensions, which is designed to minimize the distance between interdependent extensions, leaving extensions with no dependency at all at the end of the list, enforcing appearance ordering in [39]. As for checking the uniqueness of distinguished names, this fixed order also allows to easily check that each standard extension appears at most once in a digital certificate, hereby addressing Issue 6.5. After standard extensions, our format allows to specify custom extensions, maintaining the degree of freedom present in X.509 v3: such extensions, generated by the ⟨CUSTOM_EXTN⟩ nonterminal, must appear after standard ones.

Finally, the signature field concludes the certificate and contains the signature algorithm OID and the signature itself. Note that there are no algorithm parameters in signature field since we argue that it is sufficient to represent them once alongside the public key employed to verify the signature. Moreover, note that signature algorithm appears only once effectively solving the issue of context-sensitive checks reported in Issue 6.2.

We note that, despite our proposed format is missing some fields with respect to the current X.509 v3, we are able to represent with it all of the information contained in the older format, while retaining an easier to recognize, and mechanically generate, format. The only semantic constraint of X.509 v3 which we are not able to check syntactically is the uniqueness of certificate policies, custom extensions and custom distinguished names OIDs. We note that such a shortcoming cannot be coped with using a regular grammar, unless restrictions are imposed on the set of OIDs. However, we note that performing a simple bytewise uniqueness check after the recognition of the entire certificate is quite straightforward, and not likely to be mis-implemented.

Statement 6.2 (Regular syntax grammar)**.** *Analogously to the procedure employed to prove our lexical grammar regular, we can prove our syntax grammar to be regular as all of its nonterminals either generate finite languages, or are combined together with operations over which regular languages are closed. Thus, the combination of the lexical and syntactic grammars is also regular.*

6.3 Implementation strategies and experimental validation

We realized an implementation of the grammar described in the previous section, and we derived automatically a certificate parser from it. Our purpose was twofold: first of all we validated the absence of ambiguities in it, and then we experimentally validated the linear parsing time in the size of the input achieved for our new digital certificates.

Parser implementation

Our implementation employs the widely consolidated yacc toolchain [145], composed by a lexical scanner generator, `Flex`, and an LR(k) parser generator, GNU `Bison`. Our choice of a DPDA recognizer was made to provide a more meaningful error reporting, however we note that the implementation and generation of a fully regular recognizer is also possible. In implementing the lexical recognizer, we exploited the multi-state lexer generation feature available in `Flex`, which basically implement the inner product of Finite State Automata (FSAs). Such a feature allows to describe with ease the distinction between the recognition of payload and type identifiers, partitioning the inner FSA graph. The incoming byte is thus correctly matched depending on the specified lexer state. The multi-state feature is also employed to recognize constraints on public key algorithms and *key usage/extended key usage* extensions. Such constraints are different depending on the public key algorithm recognized which is matched right before them, and a solution relying on lexer states allows a clear specification of their recognition, taking into account the correct set of constraints. We

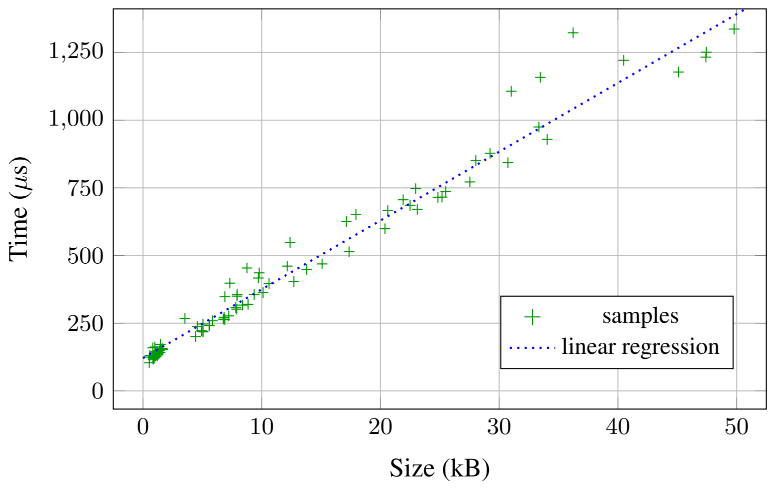


Figure 6.3: *Parsing execution timings as a function of the certificate size*

lexically check such constraints as this can be done more easily inspecting the lexical grammar terminals than inspecting whole tokens. Due to the abundance of such constraints, the lexer specification is relatively large, counting 815 lines.

However, the syntax grammar in `Bison` benefits from such an offloading, being only 457 lines long, and counting just 112 rules. Bison manages to generate the parser without conflicts, proving that the proposed grammar is unambiguous as it fulfills the LR(k) condition [70].

Performance Analysis

In order to test performance of the generated parser, we need to generate a dataset using a string generator from the same grammar specification. Such generator works in two main phases: first, it parses a yacc-formatted file to build an Abstract Syntax Tree (AST) expanding at least once every grammar production. During generation phase, the algorithm randomly traverses the tree appending strings generated by each terminal symbol found. Although the algorithm is quite straightforward, we needed to implement a new generator from scratch as the current state of the art one, Yagg [40], was not fulfilling our needs. Indeed, although Yagg allows to generate all possible strings of given length from a grammar, it has no support for binary alphabet grammars and for multi-states lexers, and it is only able to generate tokens out of a limited set of regular expression operators. Instead, our tool copes with our multi-state lexical grammar, deals with a broader set of regular expression operators and, differently from Yagg, it requires slight modifications to the `Flex`/`Bison` specifications to employ them as input.

We measured the execution times of our parser on a generated dataset made of 90 sample certificates of different lengths. The machine employed for all these tests was a Linux Gentoo 13.0 amd64 host based on a six-core Intel Xeon E5-2603v3 endowed with 32 GiB DDR-4 DRAM. The results are reported in Fig. 6.3. The good fit between the experimentally measured timing samples and their linear regression clearly shows a linear parsing time. Moreover, we note that our parser recognition times are actually really short, as recognizing a certificate of size $1\ kB$ takes only $25\ \mu$s, after a fixed bootstrapping time of $121\ \mu$s. To provide a somewhat fair comparison with existing parsers for X.509 v3 certificate, we checked the execution time of the OpenSSL [117] `d2i_X.509_fp` function, which simply reads an ASN.1 DER encoded certificate from a file and stores the information in an internal data structure, leaving the check of syntactic constraints to the validation routine. In particular, we compared the average execution time of our parser (stripped and compiled with release-grade optimizations) and the OpenSSL routine over datasets with an average certificate size of approximately 1500 bytes. The experimental evaluation showed that our complete parsing routine is as fast as the OpenSSL simpler function, since it employs 127μs on average to parse a certificate against the 126μs required by OpenSSL routine.

CHAPTER 7

Security Audit of OpenPGP Format

In this chapter, we provide our analysis of the OpenPGP format, which is extensively described in Sec. 5.3. Our analysis aims at discussing the issues and the hindrances to be tackled in order to obtain a grammar amenable to automatic parser generation algorithms for the OpenPGP format. To this extent, we first classify the set of valid OpenPGP certificates and messages according to Chomsky's hierarchy [70], showing that the OpenPGP format can be represented with a DCF grammar, from which an efficient parser can be automatically derived; then, we show that such grammar cannot be employed in practice to automatically generate an efficient parser. In addition, we point out some flaws in the specification of the OpenPGP format identified throughout our analysis, designing several attacks stemming from the exploitation of such flaws. We also evaluate the effectiveness of our attacks on the most common OpenPGP implementations, namely GPG and Symantec PGP.

7.1 On the Design of a Formal Grammar for the OpenPGP Format

We now discuss the challenges and the feasibility to design a grammar amenable to automatic parsing generation algorithms for the OpenPGP format. As a stepping stone to build such grammar, we first classify the set of valid OpenPGP certificates and messages according to Chomsky's hierarchy.

Language-Theoretical Classification of OpenPGP Format

Following the blueprint of the description of the OpenPGP format reported in Sec. 5.3, we start our analysis from OpenPGP packets, and then we classify according to Chomsky's hierarchy the set of OpenPGP certificates and messages by considering them as a language defined over OpenPGP packets. Indeed, similarly to strings in a language being a concatenation of symbols of an alphabet, OpenPGP certificates and messages are actually a concatenation of different OpenPGP packets.

OpenPGP Packets. Every OpenPGP packet has the same structure, composed by a single byte *packet tag*, a *body length* and the payload of the packet. The information found in the *packet tag* can be easily recognized by a FSA, which may simply employ up to 256 states to store the bits found in the *packet tag*. Matching the content of the *body length* field with the actual size of the payload can be also performed by a REG recognizer, as the size of an OpenPGP packet is upper bounded by $2^{32} - 1$ bytes. An exception to this upper bound is represented by the v3 packets with unknown length: for these packets, the standard delegates to implementations how to determine the end of the packet. This poses a serious issue in classifying the OpenPGP format language. Indeed, depending on the method employed to determine the end of the packet, the language can be classified in the REG class, in the DCF class (if a parenthesized structure is used), or even within classes requiring higher computational power. Given that such a choice is completely implementation dependent, in our classification of the OpenPGP object formats we will assume this encoding to be REG. Luckily, this issue was addressed in OpenPGP v4 packets, where the partial body length encoding (see Sec. 5.3) is employed for packets with unknown length; with such encoding, an arbitrarily long packet is split in several packets with at most 2^{30} bytes, in turn allowing to enforce the correct length of the packets with a REG recognizer.

The structure of the payload of an OpenPGP packet obviously depends on the packet type. Among all the packets defined in the OpenPGP standard [29], the one with the most complex structure is definitely the `Signature Pkt`, described in Sec. 5.3, because of the presence of several sub-packets. Analyzing the structure of the payload of a `Signature Pkt`, we observe that it is a sequence of elements either with a fixed size or whose size is prefixed to them (e.g., the sub-packets). In the latter case, there are upper bounds on those size: indeed, a sequence of sub-packets cannot be longer than $2^{16} - 1$ bytes, as a 16 bit wide length field, which stores the number of bytes of the sequence at hand, is prefixed to the sequence. In conclusion, to check the proper syntactical structure of a `Signature Pkt`, a REG recognizer is sufficient. Similarly, we argue that the structure of other packets appearing in an OpenPGP certificate is regular too, being simpler than `Signature Pkt` one. Indeed, all those packets are composed by elements either with a fixed size or which contain unstructured data (e.g., an encrypted message). In conclusion, since all the payloads of OpenPGP packets can be recognized with a REG recognizer, we can make the following statement:

Statement 7.1. *A generic OpenPGP packet can described with a REG language, since it is a concatenation of finite (every body length field has an upper bound) REG fields.*

OpenPGP Certificates and Messages. We can now classify according to Chomsky's hierarchy the set of OpenPGP certificates and messages by looking at their corresponding grammars, reported in Fig. 5.3(a) and Fig. 5.3(b), respectively. The terminal symbols of these grammars are OpenPGP packets, which are strings of a REG language. By looking at the grammar defining an OpenPGP certificate (Fig. 5.3(a)), we observe that it is defined by a regular expression (i.e., a single EBNF rule) over OpenPGP packets; therefore, we can make the following statement:

Statement 7.2. *The language of OpenPGP certificates is REG, as it is realized via* *union and concatenation of REG languages*

Considering the grammar defining an OpenPGP message (Fig. 5.3(b)), we observe that the production expanding the *Signed* non-terminal symbol is not linear, as it interleaves terminal symbols (i.e., `OnePassSignature` and `Signature`) and non-terminal ones (i.e, *PGPMessage*), hereby making the grammar in Fig. 5.3(b) a CF grammar. It is worth noting that any possible rewriting of the grammar in Fig. 5.3(b) cannot guarantee the non-interleaving between terminal and non-terminal symbols in the RHS of the productions. This can be easily proven observing that the OpenPGP format admits language words having a `One-Pass Signature Pkt` as a prefix and the `Signature Pkt` as a suffix. Assuming to rewrite the grammar, changing the last production and moving the non-terminal symbol *PGPMessage* either at the start or at the end of the RHS of the production, the aforementioned words can no longer be derived.

To show that the OpenPGP message format can be described by a DCF grammar, we observe that such messages exhibit the following structure: $x\,c\,b^n$ s.t. $x \in \{a, b\}^* \wedge |x|_a{=}n,\ n{\geq}0$, where a is `OnePassSignature`, b is `Signature` and c is either `Plaintext` or *Encrypted*; the number of a in x (denoted as $|x|_a$) must be equal to the number of b after c. Syntactically checking that a signed OpenPGP message exhibits the aforementioned structure requires a DPDA, thus making the grammar a DCF one.

Statement 7.3. *The OpenPGP message format is a DCF language due to signed messages employing the* single pass signature verification *format, which requires to match any* `One-Pass Signature Pkt` *found before the signed message with a corresponding* `Signature Pkt` *found after the signed message.*

In conclusion, we claim that there exists a DCF grammar that generates all the possible OpenPGP certificates and messages. Since DCF grammars are amenable to automatic parser generation algorithms, then designing a DCF grammar for the OpenPGP format would allow to immediately obtain an efficient and effective parser for OpenPGP certificates and messages with sound correctness guarantees. Nonetheless, in the following we show that, although a DCF grammar for OpenPGP format exists, it cannot be practically employed to derive a parser, because the automatically generated parser would require an unpractical amount of memory.

Size-bounds of a Parser Derived from OpenPGP DCF Grammar

The most troublesome portion of the DCF grammar for the OpenPGP format concerns the match between the content of the body length fields in OpenPGP packets with the actual number of bytes found in the payload of the packet at hand. To this end, we can employ a DPDA with a number of states which is proportional to the bit size of the body length field. Specifically, the DPDA reads the content of the body length field, pushes its binary encoding onto the stack (with the least significant bit on the top) and decrements it each time a byte of the body of the packet is read, while checking that the binary counter on the stack is zero if and only if all octets of the body of the packets have been read. Nonetheless, similarly to Issue 6.1 described for X.509 digital certificates, this strategy cannot be employed on a DPDA if more than one counter has to be simultaneously handled, since to modify one counter we erase the information on the other counter. Unfortunately, in the OpenPGP format, there is a packet whose parsing requires to simultaneously handle three counters: the `Signature Pkt`. Indeed, we recall from Sec. 5.3 that this packet contains two sequences of sub-packets: each sequence is prefixed with a 2-bytes length field, which stores the total number l of bytes encoding the sequence; furthermore, the i-th sub-packet in the sequence stores the number l_i of bytes encoding its payload in the body length field of its header. Therefore, for each byte read in the payload of a sub-packet, the DPDA should decrement the counter for the body length field of the sub-packet, the counter for the length field of the sequence the sub-packet belongs to and the counter for the body length field of the `Signature Pkt` where the sequence of sub-packets is located.

Given that the strategy of storing the binary encoding of the counters in the stack of the DPDA is not possible, then the automaton must necessarily employ the states to store these counters. We now show that the number of such states becomes so high that the automaton cannot even be deployed on existing hardware. Suppose that the DPDA employs a counter on the stack to check that the length of the sub-packet matches with the content of its body length field. Then, assuming a sequence with m sub-packets, the DPDA necessarily employs its states to verify the constraint that $l = \sum_{i=1}^{m} l_i$. In particular, the automaton needs to employ a sequence of m states for each possible "composition" of sub-packet lengths: i.e., a sequence of states for each possible list of numerical values l_i adding up to the value l. The same set of l_i values in a different order represents a different sub-packet sequence, thus requiring its own sequence of states. We now show that the number of these sequences needed for given values of m and l can be computed as $\binom{l-1}{m-1}$. Consider the numerical value l as the addition of the value one repeated l times: i.e., l=1+1+...+1. In this expression there are $l-1$ plus operators. By choosing $m-1$ plus operators to be replaced with a separator (e.g., a comma ,), we obtain m positive integers (i.e., the sum of all the 1s found between two commas) that sums up to l. We observe that there is an onto mapping between the possible placements of the $m-1$ commas and the possible set of m integers summing up to l. Therefore, the number $\binom{l-1}{m-1}$ of possible placements of $m-1$ commas over the $l-1$ available positions (i.e., the plus operators in l=1+1+...+1) corresponds to the number of possible set of integers m summing up to l. Employing m=26 and l=100, where 26 is the number of different sub-packets defined in [29], while 100 is a reasonably representative value of the size of the sequence of sub-packets, we obtain that the DPDA requires $\binom{l-1}{m-1}=\binom{99}{25}>2^{79}$ sequences of states, which implies that the DPDA memory largely exceeds the amount of memory currently available on existing hardware. Furthermore, we remark that this estimation is only an extremely loose lower bound on the number of states of the DPDA, as we should also must take into account all the possible combinations of m and l, requiring $\binom{l-1}{m-1}$ sequences of m states for each combination. In conclusion, it is not possible in practice to automatically derive a parser from the DCF grammar for the OpenPGP format.

7.2 Design Flaws in OpenPGP Format

In this section we outline the flaws of the OpenPGP format identified throughout our analysis and the attacks that exploit such flaws. To this extent, our threat model assumes an attacker to be a legitimate user of the OpenPGP system, and thus able to modify at will his keyring and to retrieve from the global keyservers any user certificate, tamper with it and upload it back. In addition, he is assumed to be able to eavesdrop OpenPGP messages sent from a legitimate user to another with the purpose of subverting the security services provided by the system.

Design Flaws and Attacks. The first issue we identify in the OpenPGP format specification is the unbounded length form of ver. 3 packets, which delegates to the application the choice of the method to determine the end of a packet encoding. Assuming a disambiguation strategy based on the presence of the End Of File marker as packet terminator, an attack aimed at both erasing and preventing the addition of subkeys to an existing certificate object is easily conceivable.

Attack 7.1. *Given a ver.* 3 *user certificate, the attacker appends to it an* `User ID Pkt` *having in its header an unbounded length specification of its body, and later exports the tampered certificate back to the keyserver. Since this packet may contain arbitrarily long unformatted data, all the packets following it in a publicly available instance of an OpenPGP certificate will be misinterpreted as part of its payload, without causing a syntactic failure.*

Note that we consider `User ID Pkt` since it is the only packet in a certificate which may contain arbitrary long unformatted data, thus subsequent packets may be enclosed in its payload without causing a syntactic failure. Fortunately, such a troublesome indefinite length encoding is discarded in ver. 4 packets. It is clear that attack 7.1 is not applicable to such an encoding, since all the bytes are still counted and a packet cannot end with a chunk having a partial body length field.

The second issue we identify lies on the fact that the specification about the signed message format with the *single pass verification* option [29] does not mandate a match between the data in the `One-Pass Signature Pkt` and the one in the corresponding `Signature Pkt`. Enforcing such a match is critical, as the content of the `One-Pass Signature Pkt` is not cryptographically signed and any alteration of the signature parameters included in it (e.g., the kind of signature algorithm) would not be spotted by a signature verification failure.

Attack 7.2. *Given a signed message including a* `One-Pass Signature Pkt`*, an attacker can replace the content of its fields without causing a syntactic failure in the OpenPGP client. As a consequence the application logic can be deceived to employ different parameters for the signature verification algorithm.*

As an example, one of the fields of the mentioned packet, namely the *signature type* one, allows to specify if the signature is to be computed on a textual or binary message. In the processing of a textual message to compute the signature, the OpenPGP specification [29] mandates to replace any octet having the same encoding of the ASCII characters $\langle\texttt{CR}\rangle$ or $\langle\texttt{LF}\rangle$ with the sequence: $\langle\texttt{CR}\rangle\langle\texttt{LF}\rangle$. Let us consider a message encoded as a binary file, and included as such in a signed OpenPGP message with a *single pass verification* option. If the attacker changes the value of the *signature type* field from binary to textual and removes one octet, for any pair of octets of the original message equal to $\langle\texttt{CR}\rangle\langle\texttt{LF}\rangle$, then the OpenPGP client will add back the removed octet during the signature verification execution, thus validating the tampered message. The consequences of deceiving the signature verification check logic depend on the semantic of the binary encoding of the original message. For instance, if the binary message contains software updates, the program code may be tampered with this technique.

The third issue we identify is related to the structure of the `Signature Pkt`. In particular, the two sequences S_1, S_2 of sub-packets in the body of a `Signature Pkt` may include sub-packets of any type, with the ones in S_1 being signed, while the ones in S_2 being unsigned (see Sec. 5.3). The OpenPGP standard [29] does not report any restriction nor recommendation on which kind of sub-packets should appear in S_1 or S_2, potentially exposing applications to malicious information found in the unsigned sub-packets. Indeed, an implementer of the OpenPGP clients may not have enough expertise to determine whether an information found in an un-signed sub-packet may be employed without consequences on the security guarantees. We now describe two attacks on two different sub-packet types that show possible consequences of trusting sensitive information found in an un-signed sub-packet.

Attack 7.3. *Given a user's certificate with a self-signature on the UserID/Primary Public Key binding, the attacker adds to the portion encoded by the* `Signature Pkt` *an unsigned* `Revocation Key Subpkt`. *This sub-packet includes the digest of a cryptographic key meant to be able to revoke the binding vouched by the signature in the* `Signature Pkt`. *In case the OpenPGP client processes a certificate taking the information in an unsigned* `Revocation Key Subpkt` *sub-packet as legit, an attacker is able to revoke the original UserID/Primary key binding with the key included in the added sub-packet*

Furthermore, the preferred algorithms subpackets, that are `Preferred Hash Algorithms Subpkt`, `Preferred Compression Algorithms Subpkt`, `Preferred Symm Algorithms Subpkt`, may be exploited to perform a downgrade attack:

Attack 7.4. *Given a user's certificate with a self-signature on the UserID/Primary Public Key binding, the attacker adds to the portion encoded by the* `Signature Pkt` *an unsigned sub-packet to state the cryptographic algorithms to be preferably used to interact with the UserID. The possible subpackets of this kind are:* `Preferred Hash Algorithms Subpkt`, `Preferred Symmetric Algorithms Subpkt` *and* `Preferred Compression Algorithms Subpkt`. *In case the OpenPGP client processes a certificate taking the information in an unsigned sub-packet as legit, an attacker is able to foster the usage of weak cryptographic primitives to communicate with the victim user.*

For instance, by stating that a user supports only MD2 algorithm, the attacker forces the usage of a weak hashing primitive where collisions may be found.

A further issue is related to the lack of standardized criteria for managing conflicting information in OpenPGP certificates and messages containing `Signature Pkts`. In particular, there are two possible conflicting scenarios. In the first one, the lists of sub-packets in a `Signature Pkt` may contain one or more sub-packets of the same type with conflicting data. In the second one, a OpenPGP certificate may include multiple self-signatures vouching for the same binding, but including conflicting data. Non-standardized strategies for solving the aforementioned conflicts are a potential source of vulnerability. Indeed, if two different implementations employ different policies, then an attacker can benefit from the different data processing, as we exemplify in the following attack.

Attack 7.5. *Given a user's certificate with a self-signature including an expiration time t_1 in a* `Key Expiration Time Subpkt`*, the attacker may add an additional unsigned sub-packet of the same type with expiration time $t_2 > t_1$ with the aim to impersonate the certificate's owner in the future after recovering its private key. The conflicting information on the expiration date may lead a set of clients to consider the certificate valid up to t_2, while the user still believes its key to be valid until t_1.*

7.3 Experimental Evaluation of Our Attacks

We now evaluate the attacks identified in Sec. 7.2 against both Symantec PGP [148] 10.4.0MP1HF1 and GPG [89]. For GPG, we test both the legacy tree version $ver.$1.4.18, employed by default in Debian and Ubuntu, and ver. 2.1.18. However, we did not find any differences, and thus we will refer to both of them as GPG from now on. As a keyserver, we employ SKS [110], the de-facto standard implementation written in OCaml, which is currently used in the overwhelming majority of the public keyservers.

Attack 7.1 Evaluation. This attack is simply prevented in GPG since a `User-ID Pkt` with an unbounded length type of *body length* field is deemed a syntactic error. We observe that the attack conceptually works through usage of `Literal Pkt`, on which GPG allows usage of unbounded length type of *body length* field. Indeed, if we inject a `Literal Pkt` in an OpenPGP certificate, all the subsequent packets will be included in its body. Nevertheless, this certificate is not even exported by GPG since a `Literal Pkt` is valid only if found in OpenPGP messages. Moreover, even directly uploading the certificate to our test keyserver, we found out that it is ignored by SKS. Indeed, by looking at the code, we realize it does not handle unbounded length encoding at all. About Symantec PGP, no error is raised during the import of the certificate, however the unbounded length `User-ID Pkt` is ignored, with no effects on the original certificate.

Attack 7.2 Evaluation. Signature type and public key algorithm identifier fields in `One-Pass Signature Pkt` are ignored by GPG, while a mismatch between the hash algorithm identifiers in `One-Pass Signature Pkt` and in the corresponding `Signature Pkt` is detected. Symantec PGP detects the mismatch for each of the previous fields. Therefore, Attack 7.2 is not feasible on both implementations.

Attack 7.3 Evaluation. First of all, we acknowledge that the modified self signature with the unsigned `Revocation Key Subpkt` is imported in the keyring by GPG only if the original, untampered signature is absent in the keyring. However, the unsigned sub-packet is ignored even if imported in the keyring. Symantec PGP is even more restrictive, completely preventing to import the modified self signature with the unsigned `Revocation Key Subpkt` in the keyring independently from the fact that the original, untampered signature is already present in the keyring or not.

Attack 7.4 Evaluation. This attack does not work since both implementations include by default signed preferred algorithms sub-packets. These ones state legitimate preferences to be used when interacting with the UserID authenticated in the `Signature Pkt`. These preferences cannot be restricted employing another preferred algorithm sub-packet, but they can only be enlarged. Therefore, the attacker cannot force the usage of a weak cryptographic primitive.

Attack 7.5 Evaluation. For the same reasons highlighted during analysis of attack 7.3, this one can be performed only against users that do not have the victim's public key in their keyring. GPG employs a signed `Key Expiration Time Subpkt` to state when the public key expires. Therefore, the unsigned `Key Expiration Time Subpkt` is ignored. Since Symantec PGP ignores all added sub-packets, it is clearly immune to this attack too.

As a conclusion, none of the identified attacks are exploitable on considered implementations. We remark that this evaluation does not show that these issues are harmless, but it demonstrates that fortunately implementers properly deal with them, being conscious of the security implications of their decisions.

7.4 Improvements to OpenPGP Format

We now briefly discuss some modifications to the OpenPGP format to obtain a DCF grammar that can be employed in practice to derive an automatically generated parser.

The presence of the *single pass signature verification* option in the format of the signed messages is responsible for the DCF classification of the OpenPGP grammar. A change in the grammar definition to make the format amenable to a REG specification would require an additional constraint on the maximum number of different signatures put on the same message, removing the need of a recursive production in the grammar. However, such a modification would reduce the expressiveness of the format language and in the end remove a feature of the OpenPGP system.

The specification of a practically manageable OpenPGP DCF grammar would require to remove the length fields from OpenPGP packets. We note that in the current specification of the OpenPGP format, some length fields are redundant (e.g., when referring to a fixed sized sequence of elements). In such cases they can be safely removed to keep the format as small as possible. To remove the non redundant length fields, similarly to the approach employed in the design of our novel format for X.509 digital certificates described in Chapter 6, we may replace length fields with proper delimiters, say a fixed octet, provided a proper escaping is computed over the octets in case the same value of the delimiter octet is found in the payload of the packet. This escaping can be easily performed by doubling the delimiter octet in encoding, and removing it in decoding. Therefore, the end of the element is found when an odd number of delimiter octets is matched. We observe that such new binary encoding for OpenPGP packets cannot completely replace the older one: indeed, because of the append-only nature of OpenPGP certificates, it is not possible to get rid of old packets in existing OpenPGP certificates; nonetheless, it would be possible to append packets without length fields to existing OpenPGP certificates. Conversely, OpenPGP messages can be entirely encoded without length fields.

Finally, we discuss mitigation strategies for each of the attacks identified in Sec. 7.2. Attack 7.1 is fortunately already mitigated in ver. 4 OpenPGP packets. Therefore, existing implementations, while still accepting valid ver 3 packets for backward compatibility, should foster the adoption of ver. 4 ones by employing them while creating or updating OpenPGP certificates and messages. Nonetheless, our experimental validation reports that both GPG and Symantec PGP still generate certificates and messages employing ver. 3 packets. Attack 7.2 can be mitigated by mandating that the parameters for the computation of the digital signature found in a `One-Pass Signature`

`Pkt`, which are unsigned, must match with the corresponding ones found in the signed `Signature Pkt` paired with the `One-Pass Signature Pkt` at hand, as currently enforced by Symantec PGP. Attacks 7.3, 7.4 are caused by both the append-tolerant nature of the OpenPGP format (which is crucial to its working) and the freedom of action left by the standard on the decision of which sub-packets should be signed. Since removing the possibility to append valid packets to existing OpenPGP objects would undermine the functionality of the system, the only viable alternative is forcing all the security critical sub-packets (e.g., `Non-Revocable`, `Key Expiration Time`, `Revocation Key`, and `Signature Expiration Time`) to be signed. Attack 7.5 is caused by a loose specification in the standard on how to deal with conflicting information found in OpenPGP certificates and messages, which may lead to inconsistencies of the data extracted from an OpenPGP certificate or message between different implementations. Therefore, to remove this potential threat, the OpenPGP standard should mandate a specific policy to handle conflicting information, such as introducing uniqueness constraints or always considering the first/last duplicate information processed.

Concluding Remarks and Further Developments

The proper adoption of sound cryptographic primitives is crucial in many real world systems to ensure their security: indeed, even a single weakness found in a cryptographic component of the system is often sufficient to completely subvert its security guarantees. Unfortunately, employing theoretically sound cryptographic primitives is not sufficient to ensure that the cryptographic components of a system are not exploitable by attackers. Indeed, the integration and deployment of a cryptographic primitive in a complex system usually exhibits some challenges that must be carefully tackled by designers and developers. For instance, given the complexity of cryptographic primitives, developers may easily build flawed implementations, where a simple unnoticed error may introduce algebraic or statistical weaknesses as well as making the implementation vulnerable to powerful side-channel attacks. Furthermore, even assuming the adoption of a sound implementation from a reliable cryptographic library, other potential vulnerabilities may be introduced by misusing cryptographic components (e.g., employing a weak secret key). Besides ensuring that the cryptographic components provide the expected security guarantees, the designers of a secure system are also concerned by the performance overhead introduced by the cryptographic components. Indeed, in case a cryptographic primitive significantly degrades the performance of a system, it may even be removed from the system, hereby leaving it with weaker or even no security guarantees, as no designer would employ a cryptographic solution that significantly worsens the quality and the usability of the system.

In this work, we addressed the challenges imposed by the adoption of two cryptographic primitives in real-world systems and applications. Specifically, we focused on the security cost exhibited by solutions for privacy-preserving outsourced computation and on the exploitable security vulnerabilities introduced by the improper parsing of digital certificates. For each of these broad topics, which correspond to the two parts composing this manuscript, we now briefly recap the specific problems tackled in this work, our contributions and the conclusions that we can derive from our research. We also discuss possible further developments of our work that may stem from our findings.

Privacy-Preserving Outsourced Computation

We focused on the problem of outsourcing computation to a powerful untrusted server (e.g., a cloud provider or a data center) while retaining the confidentiality of the data involved in the computation. Indeed, the lack of privacy guarantees on outsourced data may prevent entities dealing with sensitive data (e.g., biomedical or financial ones) to outsource computation to an untrusted server, in turn leading to the loss of all the financial and performance advantages enabled by outsourcing computation. FHE schemes allow to perform an arbitrary computation over encrypted data without the need to know the secret key, thus guaranteeing the confidentiality of the data involved in an outsourced computation. The main hindrance that prevents the adoption of FHE schemes in real-world applications is the significant performance overhead introduced by computation over ciphertexts, which amounts to at least 6 order of magnitudes with respect to standard computation over the corresponding plaintext values.

The first approach followed in this work to reduce this performance gap was the investigation of the security guarantees of noise-free FHE schemes: indeed, these schemes can in principle exhibit better performance than common FHE ones because of the absence of costly noise management techniques; nonetheless, most of the existing noise-free schemes turned out to be completely insecure after some scrutiny. Our investigation lead to the design of new attack techniques against FHE schemes (reported in Chapter 2), which are applicable also to the only two existing noise-free ones that had not been completely broken yet (i.e., `OctoM` and `JordanM`). Our main technique is a plaintext-recovery attack applicable to any FHE scheme for which there exists an m-distinguisher, that is an efficient algorithm capable of determining if the corresponding plaintext value of a generic ciphertext is a fixed integer m. Our attack combines the homomorphic capabilities of the FHE scheme with the information leakage coming from the m-distinguisher, employing only the public information about the ciphertexts available in a ciphertext-only scenario. Although the computational cost of our attack is linear in the value of the plaintext being recovered, it significantly improves the number of recoverable plaintexts with respect to an exhaustive search strategy, which, in turn, might mean recovering a vast portion of ciphertexts in a FHE application scenario. Since the existence of an m-distinguisher was proved for any linearly-decryptable FHE scheme [158], our attack is immediately applicable to any such scheme, including the two linearly-decryptable noise-free FHE schemes `OctoM` and `JordanM`. We indeed experimentally validated the efficacy of our attack against these two FHE schemes, also showing that the performance of a parallel implementation of our attack scales proportionally with the number of computing nodes employed to perform the attack. Furthermore, during the implementation of the `OctoM` scheme, needed for our experimental validation, we discovered that it is not a FHE scheme, because the homomorphic multiplication proposed by its designers in [158] yielded an incorrect result; therefore, we also revised this homomorphic operation in `OctoM`, proposing a sound design that makes the scheme fully homomorphic.

In addition, we discussed a simple modification to `OctoM` and `JordanM` that makes them no longer linearly-decryptable, and thus no longer vulnerable to our plaintext recovery attack; nonetheless, we showed that it is still possible to mount a KPA with the knowledge of the corresponding plaintext value for a few tens of ciphertext, and we proposed a second attack technique that allows to compute enough ciphertexts with known corresponding plaintext value to mount the KPA. This second technique allows to compute a polynomial number of ciphertexts with known plaintext values for any FHE scheme by relying on a very little amount of information, namely only a broad range of possible plaintext values for a single ciphertext is sufficient. We showed that this limited information allows to efficiently compute enough ciphertexts with known plaintext values to mount the KPA against the modified `OctoM` and `JordanM` schemes, in turn completely subverting their confidentiality guarantees.

Our attacks clearly show that FHE schemes must ensure strong security guarantees (i.e., at least security against *chosen-plaintext* attacks), as otherwise they are completely insecure. Indeed, the common thread of our techniques is that the homomorphic capabilities allow to dramatically amplify the impact of a small vulnerability on the security of the vulnerable FHE scheme. Our investigation unfortunately poses a further challenge in the design of secure and efficient FHE schemes, since it implies that we cannot trade strong security guarantees of the scheme for the efficiency of the homomorphic computation. For instance, the `OctoM` and `JordanM` noise-free FHE schemes were an example of this trade-off, as the existence of both the m-distinguisher and the KPA, acknowledged also by their authors, might still have allowed the adoption of these schemes in application scenarios where it is safe to assume that the attacker cannot know enough plaintext values to mount the KPA or where it is unlikely to find ciphertexts whose corresponding plaintext is the fixed value m; nonetheless, our attacks show that these schemes cannot be securely used even in such particular scenarios. Therefore, the design of secure and efficient noise-free FHE scheme remains an open problem, which may be addressed in future research efforts.

Given the absence of a secure noise-free FHE scheme, in this work we investigated two alternative strategies to overcome the performance gap introduced by FHE: employing PHE schemes instead of FHE ones, as they trade the homomorphic capabilities for efficiency; relying on trusted hardware for secure computing, such as the commercially available Intel SGX technology. In the first case, the main challenge resides in devising algorithms to perform a given computation with the limited homomorphic capabilities of a PHE scheme; in the second case, the main challenge is mitigating the information leakage coming from SGX side channels. In this work, we applied these two approaches in the design of two efficient PPSS protocols which exhibit practical performance and a limited communication cost between the user issuing substring-search queries and the untrusted server that computes the results of such queries. We now recall the main achievements of our designs and the possible future developments that we foresee to improve their performance and their features; then, we discuss the effectiveness of our solutions in reducing the performance overhead of privacy-preserving outsourced computation.

In the first of our solutions, presented in Chapter 3, we provided a general construction of a PPSS protocol, which combines the privacy guarantees given by a PIR protocol with the backward search substring search method [57]. By instantiating our construction with the Lipmaa's PIR protocol [96], which is based on the LFAHE scheme by Damgård and Jurik [44], we obtained the first PPSS protocol with proven guarantees of search and access pattern privacy that enables the simultaneous execution of queries from multiple users without the need of the data owner being online, and exhibiting a sub-linear $\mathrm{O}(m\log^2(n) + o_q\log(n))$ communication cost per user. In a multi-user scenario, our protocol requires only $\mathrm{O}(\log^2(n))$ additional memory for each query simultaneously performed, thus avoiding the replication of the whole full-text index for each query, in turn improving the scalability of our solution. Furthermore, our PPSS protocol is the first one enabling privacy-preserving pattern matching over outsourced data, since it allows to perform queries containing the most common wildcard characters while leaking a limited information about the structure of the searched pattern and the matching portions of the outsourced data. The experimental validation of our PPSS protocol over a real-world genomic use case showed practical response times, requiring few minutes on off-the-shelf hardware to compute the results of a query over 21^{st} human chromosome, which contains about 40 MiB of genetic data. Remarkably, our PPSS protocol requires an extremely low bandwidth, which amounts to less than 50 KiB for each of the $\mathrm{O}(m)$ communication rounds; this limited bandwidth makes our solution applicable also to scenarios where the end user issuing a query may reside on a device with a low latency connection and a non flat-rate plan for network access (e.g., a mobile phone), which was one of the requirements we initially established for our PPSS protocol.

Despite the practicality of the response time achieved by our solution, the computational effort of the untrusted server is definitely the bottleneck in our PPSS protocol. A possible solution to improve the overall response time of our protocol may be instantiating our generic construction with a different PIR protocol. In particular, two recently proposed PIR solutions (XPIR [109] and SealPIR [5]) showed significant improvements in terms of actual execution time at server side by relying on lattice-based AHE schemes instead of number-theoretic ones such as the DJ LFAHE scheme employed in this work. Nonetheless, both these solutions are based on the PIR protocol proposed by Stern in [146], whose communication cost is $d \cdot n^{\frac{1}{d}} + F^d$, where $F = \Theta(1)$ and d is a value chosen by the client at each query. We observe that these protocols cannot achieve a poly-logarithmic communication cost: indeed, if $d = \mathrm{O}(1)$, then the communication cost is $\mathrm{O}(n^{\frac{1}{\mathrm{O}(1)}})$; if $d = \Theta(\log(n))$, then the communication cost becomes $\mathrm{O}(\log(n) + F^{\log(n)}) = \mathrm{O}(n)$. Therefore, in this work we did not consider these solutions, given our goal of minimizing the communication cost with the extent of making our protocol usable in scenarios with limited bandwidth (e.g., on mobile phones); nonetheless, these two solutions may be employed in a future work to explore a better trade-off between the bandwidth and the computational effort at server side, with the extent of minimizing the overall response time for the queries of our PPSS protocol. In addition, we remark that since any PIR protocol can be employed to instantiate our construction, any improvement in terms of computation or communication in a PIR solution may be immediately applicable to our construction, thus obtaining a PPSS protocol with similar performance improvements.

To asymptotically reduce both the communication and the computational costs, a possible solution is the construction of a multi-user PPSS protocol by relying on an ORAM protocol. In particular, we aim at employing PanORAMa [121], a recently proposed ORAM construction that achieves a $\mathrm{O}(\log(n)\log\log n)$ (amortized) communication cost. Nonetheless, a well-known limitation of ORAM protocols is the difficulty of simultaneously perform accesses to the ORAM data structure, which is mainly due to the stateful nature of ORAMs. Therefore, in order to employ an ORAM protocol in place of a PIR one in our multi-user PPSS solution, we need to overcome this limitation of ORAMs; a possible solution may be adapting to PanORAMa the constructions found in [160, 32], which allow to simultaneously perform accesses to the same ORAM for multiple users. Nonetheless, these constructions do not take into account the information leakage among different users, that is these solutions do not aim at guaranteeing the privacy of the access pattern of a user against the other users accessing the ORAM. Since this feature makes our PPSS protocol resistant against collusion among the untrusted server and authorized users, it would be interesting to address also this challenge, thus obtaining a solution with the same security guarantees of our construction based on PIR, but with improved computational and communication costs.

The second PPSS protocol we proposed in this work, described in Chapter 4, relies on the security guarantees of the Intel SGX technology. To reduce the information leakage coming from SGX side channels, our PPSS protocol employs an ORAM with oblivious client algorithms, referred to as DORAM. In order to find the best performing DORAM for our PPSS solution, we designed three doubly oblivious versions of existing ORAMs: our own design of Path DORAM, which improves the performance of existing doubly oblivious versions of Path ORAM employed in previous works, and the first design of Circuit and Ring DORAMs; these contributions are relevant also beyond the scope of this work, as our DORAMs may be employed to build other privacy-preserving applications based on SGX or to speed-up existing countermeasures that rely on the privacy guarantees of a DORAM to mitigate the information leakage coming from side channels in general purpose SGX applications [135, 4]. By combining our DORAMs with an oblivious version of the backward search method, we obtained ObSQRE, the first PPSS solution that guarantees search and access pattern privacy of substring search queries while exhibiting an **optimal** communication cost of $\mathrm{O}(m + o_q)$ in 1 communication round. Furthermore, ObSQRE exhibits a remarkable $\mathrm{O}((m+o_q)\log^3(n))$ computation cost for the untrusted server, which is the lowest one among existing PPSS solutions guaranteeing search and access pattern privacy. In addition, ObSQRE guarantees both the search and access pattern privacy as well as the correctness of the result of the queries even against a malicious adversary who has total control of the machine hosting the SGX enclave, improving over the security guarantees of our multi-user PPSS protocol based on PHE, which hold only against a semi-honest adversary. The experimental validation of ObSQRE showed that our solution is extremely practical for real-world use cases: indeed, it requires only about 1 second to privately retrieve all the occurrences of a protein (about 3000 nucleotides) over the entire human genome, which contains approximately 3 GiB of genetic data; the query time is reduced to few milliseconds for the retrieval of the occurrences of short keywords (i.e., less than 20 characters) in the `Enron` email dataset, which contains more than 1 GiB of emails belonging to accounts of the employees of a real-world financial firm. These experiments show that ObSQRE is already usable in real-world applications without incurring in a noticeable performance degradation.

Despite the astonishing performance and the higher security guarantees, ObSQRE lacks some important features of our PPSS protocol based on PHE: first, because of the usage of a DORAM, it does not allow to perform simultaneously multiple queries from distinct users; secondly, it does not support pattern matching queries. We have already started working to overcome these limitations. In particular, our strategy to adapt ObSQRE to a multi-user scenario relies on the approach followed by TaoStore [131], a solution that allows to simultaneously perform multiple accesses to an ORAM by relying on a proxy application trusted by all the users, which is run on a trusted machine that is assumed not to collude with the untrusted server storing the ORAM. In our scenario, we can remove this limiting non-collusion assumption by moving the proxy inside the SGX enclave, which allows also to overcome the main bottleneck that curbs the scalability of TaoStore only to few users: the network bandwidth. Nonetheless, moving the proxy inside an SGX enclave poses the significant challenge of making it oblivious and compatible with a DORAM, as otherwise the privacy guarantees of TaoStore would be subverted by the information leakage coming from SGX side channels. Our efforts are currently driven to the design of an efficient oblivious version of the TaoStore proxy, which will enable to simultaneously perform multiple queries in ObSQRE. We remark that in case of a successful design of an oblivious TaoStore proxy, we will obtain a solution that enables multiple simultaneous accesses to a DORAM, which may be of independent interest for a generic privacy-preserving application based on SGX. Finally, to enable pattern matching queries in ObSQRE, we are planning to design oblivious versions of the algorithms employed in our multi-user PPSS protocol (see Sec. 3.2).

Although in ObSQRE we hinged upon a DORAM to address the information leakage coming from SGX side channels, an alternative viable solution is represented by *dynamically allocated data structures* [51, 53], in particular by the shuffle index [50, 48, 52]. This data structure aims at the same privacy guarantees of an ORAM protocol, namely guaranteeing access pattern confidentiality to data stored in an untrusted memory, while providing richer capabilities to the client: indeed, the shuffle index, besides the retrieval of the data based on its identifier, supports efficient range queries (i.e., retrieving all the data whose identifiers belong to a given interval) and allows to insert and remove elements in the outsourced data. From a performance standpoint, the shuffle index exhibits similar response times w.r.t. Path ORAM, and, for expected choices of the parameters employed to ensure the access pattern privacy, a lower bandwidth consumption [50]. In order to employ a shuffle index in ObSQRE, we would need to make the algorithms executed at client side oblivious, as in our DORAMs. Although the operations performed at client side to access elements in the shuffle index are more complex than the ones of our DORAMs (e.g., the client would need to obliviously shuffle blocks retrieved from the index), they involve less data; thus, it would be interesting to evaluate if such an oblivious shuffle index may further improve the performance of ObSQRE. We chose to defer this investigation to future works, as one of our goals was the evaluation of different performance trade-offs between several DORAM designs, given their relevance also for other existing countermeasures to withstand side channels in general purpose SGX applications [135, 4]. Finally, we remark that a shuffle index able to efficiently deal with multiple simultaneous accesses from different users is proposed in [49], thus the adoption of this privacy-preserving data structure may be useful also to achieve our goal of making ObSQRE amenable to a multi-user scenario.

We now analyze the effectiveness of our two solutions in improving the performance of privacy-preserving outsourced computation in the considered application scenario. To this extent, we compare in Tab. 7.1 the response times of the substring-search queries in our two PPSS protocols with the one of the PPSS protocol based on FHE reported in [80]. We remark that, despite the tests were not performed on the same machine, similar CPUs were employed; furthermore, given the different magnitude of the query

Table 7.1: *Comparison of the query response time among PPSS protocols employing different privacy-preserving solutions*

Privacy-Preserving Solution	CPU	Threads	Dataset Size	Substring Length	Response Time
FHE Based [80]	Xeon E5-1620	72	0.5 MB	5	30 min
PHE Based	Xeon E5-2620	21	40 MB	6	5 min
SGX Based	Xeon E3-1220	1	3 GB	3000	1 s

response times among the solutions, it is rather clear that the results are negligibly affected by the machine employed for the experimental evaluation. The response times in Tab. 7.1 clearly shows the performance improvements achieved by our solutions: indeed, our multi-user PPSS protocol based on PHE is 1 order of magnitude faster than the FHE based solution, despite the queries are executed employing less computing units and they are performed over a document that is 2 order of magnitudes bigger; the performance gap is even bigger between our multi-user solution and ObSQRE, showing the significant performance gain given by relying on trusted hardware instead of purely cryptographic solutions. This comparison clearly shows that both our approaches are effective in achieving our goal of reducing the performance overhead of privacy-preserving outsourced computation. Nonetheless, both these approaches also exhibit some drawbacks that prevent their general adoption.

First of all, we remark that PHE can be employed only to a limited number of application scenarios, as there are some computations that can only be represented by an arithmetic circuit with both additions and multiplications. Furthermore, the weaker homomorphic capabilities of a PHE scheme make the design of the privacy-preserving solution much more challenging with respect to FHE, although the development of the privacy-preserving application is simpler as PHE does not need to handle noise growth in homomorphic operations. Conversely, the design of privacy-preserving solutions based on Intel SGX, besides yielding astonishing performance improvements, is also definitely simpler, as there is no need to represent all the computations as arithmetic circuits. Nonetheless, HE schemes still presents several advantages over SGX technology, which are mainly related to the widespread adoption of such technology in practice.

Indeed, a relevant issue to be faced for the actual deployment in the cloud of privacy-preserving solutions based on SGX is the limited availability of powerful computing infrastructures where SGX is available and enabled in the BIOS firmware of the machine. Specifically, an SGX application can obviously run only on a machine whose CPU is equipped with SGX; although there are several machines in the cloud with a CPU where SGX is available, several cloud providers keep SGX disabled on such machines. This policy is mostly due to concerns about the impact of SGX over the security of the machine hosting the enclave. Indeed, while in SGX threat model the code running in the enclave is always assumed to be trusted, a cloud provider cannot rely on this assumption for enclave applications run by one of its tenants; this discrepancy introduces a strong conflict between the privacy guarantees that SGX must ensure to enclave applications and the need for the cloud provider to profile and inspect the execution of untrusted code in order to protect the security of its infrastructure. This conflict may be solved only if the cloud provider can be reassured that the code running in the enclave has no way to harm the system thanks to the constraints imposed by enclave execution (e.g., forbidden system calls); nonetheless, it has been recently showed in [137] that it is possible to lead several attacks (e.g., stealing private files, phishing on behalf of the owner of the machine) from a malicious payload hidden inside an SGX enclave application. This work clearly showed the need to deeply analyze the security implications of running untrusted SGX applications for the machine hosting the enclaves, as well as pushing for the identification of feasible countermeasures to reduce the attack surface available for malicious SGX applications [159] without undermining the privacy guarantees of SGX. It is likely that cloud providers will be reluctant to enable SGX on their infrastructure without a solid understanding of the potential threats coming from SGX enclaves and without effective mitigations that prevent the exploitation of SGX enclaves as an attack vector; therefore, given the yet poor scrutiny of such threats, this is currently a serious hindrance to the flourishing of privacy-preserving applications based on SGX technology.

Finally, another advantage of HE techniques with respect to trusted hardware relies on the autonomy of the solution from the hardware vendor. Indeed, an HE application can be deployed on any machine, with no involvement of the vendor of the device where the computation is performed; conversely, with Intel SGX, it is possible to remotely attest an enclave (and thus trust the code and data found inside it) only with the participation of Intel. This need may limit the access to private outsourcing in some circumstances, as Intel may hinder the deployment of an SGX based application to protect its own interests. This possibility, although applicable only to peculiar scenarios, have raised a debate on the implications and ethical concerns related to the market power acquired by a single hardware vendor [42, pag. 90], and may encourage some entities (e.g., governments) to prefer the adoption of privacy-preserving techniques where they can retain more independence from the hardware vendor.

Digital Certificates

We tackled the problem of proper parsing of digital certificates, which are widely employed in several secure communication protocols to ensure the authenticity of the binding between a public key and its owner. Throughout the years, several MitM attacks against such protocols have stemmed from parsing inconsistencies found in existing parsers for digital certificates. In our work, we aimed at improving the accuracy of such parsers by automatically derive them from grammar specifications. This approach was already applied to X.509 digital certificates [13] and was showed to be really effective in improving the parsing accuracy. Nonetheless, given the complexity of the X.509 format, authors of [13] were forced to employ a large predicated grammar to describe the format, whose complex design and structure may hinder the development and maintenance of this solution in real-world implementations.

Therefore, in this work we proposed a novel format for X.509 digital certificates, which was explicitly design to address all the parsing hindrances found in the current format, while retaining the same expressiveness of existing X.509 digital certificates. The main desirable feature achieved by our new format is the existence of a regular grammar describing the format, as from this grammar it is possible to automatically derive a parser with sound correctness guarantees and with optimal parsing complexities. Furthermore, we designed our new format to ease its gradual roll-in in the PKI, as digital certificates in our own format can be recognized as ver. 4 ones by implementations recognizing only older versions of X.509 certificates; in this way, these implementations can gracefully stop parsing the certificate without incurring in potential side effects given by interpreting certificates in our new format as older versions of X.509. We consider this graceful handling of our new format by existing implementations as a key enabler to overcome the usual interoperability hindrances related to the widespread adoption of a new format; indeed, certificates in our own format can be progressively introduced in the PKI, upon expiration of the current ones, without causing disruptions in legacy systems. The experimental validation of the parser automatically derived from our grammar via the yacc toolchain showed performance comparable to existing parsing libraries for X.509 v3 digital certificates and a linear parsing time.

The next goal in our research will be automatizing the identification of parsing inconsistencies in existing recognizers of X.509 digital certificates, by designing a tool that relies on a grammar specification of the X.509 format to automatically generate a set of digital certificates that can be employed to assess the accuracy of parsing libraries. In particular, we aim at devising a mechanism to automatically inject errors in the grammar specification of the format, in order to generate a set of erroneous digital certificates which are labeled with the syntactic flaw that makes them invalid. In our vision, this labeled testset could be automatically generated after each important upgrade of an X.509 parser to automatically identify new parsing inconsistencies; furthermore, it would provide to developers a finer-grain error reporting, which surely aids them in fixing the identified parsing inconsistencies.

Besides X.509 digital certificates, in this work we also focused on the problem of properly parsing OpenPGP certificates and messages. In particular, following the same systematic approach that turned out to be effective for X.509 digital certificates, we aimed at obtaining an automatically generated parser for OpenPGP certificates and messages. To this extent, in this work we performed a deep analysis of the OpenPGP format, which proved that it can be described with a DCF grammar. Nonetheless, we also showed that, although such grammar exists, it is impossible to employ it in practice to automatically derive a DPDA parsing OpenPGP certificates and messages, as such automaton would require an amount of memory that largely exceeds the capacities of existing memory devices. We also discussed some modifications to the OpenPGP format to overcome this issue and enable the automatic generation of a DPDA to recognize OpenPGP certificates and messages. In addition, throughout our analysis we identified several flaws in the OpenPGP format specification and we outlined five attacks stemming from the exploitation of such flaws. We experimentally verified that our attacks are ineffective against the most common OpenPGP implementations, namely GPG and Symantec PGP, mostly because their developers properly applied the OpenPGP specification by bearing in mind the security implications of their implementations choices. Nonetheless, this outcome does not mean that the identified flaws are harmless, and thus we also proposed some little modifications to the OpenPGP standard that would prevent our attacks.

Given the impossibility of automatically generating a parser from a DCF grammar, we are planning to investigate other solutions to automatically derive a parser with sound correctness guarantees. Among the possible approaches, we may consider to rely either on predicated grammars [120], as it was done for X.509 digital certificates, or on *parser combinators* [9], which allow to specify a format as a set of predefined data structures, then automatically deriving a parser by composing the sub-parsers for each of these data structures. Although these approaches do not ensure the same formal guarantees of DCF grammars, they reduce the complexity of parser development and we expect them to improve the parsing accuracy with respect to existing handcrafted parsers for OpenPGP certificates and messages.

Bibliography

[1] The OpenPGP HTTP Keyserver Protocol (HKP). RFC Internet Draft, 2003.

[2] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4):79:1–79:35, 2018. doi: 10.1145/3214303. URL `https://doi.org/10.1145/3214303`.

[3] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In Ray et al. [125], pages 5–17. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813707.

[4] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. OBFUSCURO: A commodity obfuscation engine on intel SGX. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL `https://www.ndss-symposium.org/ndss-paper/obfuscuro-a-commodity-obfuscation-engine-on-intel-sgx/`.

[5] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 962–979. IEEE Computer Society, 2018. doi: 10.1109/SP.2018.00062. URL `https://doi.org/10.1109/SP.2018.00062`.

[6] John C. Baez. The octonions. *Bulletin of the American Mathematical Society*, 39(2):145–205, 2002. doi: 10.1090/S0273-0979-01-00934-X.

[7] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proc. of the 18th ACM Conf. on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 691–702. ACM, 2011. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046785. URL `https://doi.org/10.1145/2046707.2046785`.

[8] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 719–737. Springer, 2012. doi: 10.1007/978-3-642-29011-4_42. URL `https://doi.org/10.1007/978-3-642-29011-4_42`.

[9] Julian Bangert and Nickolai Zeldovich. Nail: A practical tool for parsing and generating data formats. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 615–628. USENIX Association, 2014. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert`.

[10] Alessandro Barenghi, Alessandro Di Federico, Gerardo Pelosi, and Stefano Sanfilippo. Challenging the Trustworthiness of PGP: Is the Web-of-Trust Tear-Proof? In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, volume 9326 of *LNCS*, pages 429–446. Springer, 2015. ISBN 978-3-319-24173-9.

[11] Alessandro Barenghi, Nicholas Mainardi, and Gerardo Pelosi. A security audit of the openpgp format. In *14th International Symposium on Pervasive Systems, Algorithms and Networks & 11th International Conference on Frontier of Computer Science and Technology & Third International Symposium of Creative Computing, ISPAN-FCST-ISCC 2017, Exeter, United Kingdom, June 21-23, 2017*, pages 336–343. IEEE Computer Society, 2017. doi: 10.1109/ISPAN-FCST-ISCC.2017.35. URL `https://doi.org/10.1109/ISPAN-FCST-ISCC.2017.35`.

[12] Alessandro Barenghi, Nicholas Mainardi, and Gerardo Pelosi. Comparison-based attacks against noise-free fully homomorphic encryption schemes. In David Naccache, Shouhuai Xu, Sihan Qing, Pierangela Samarati, Gregory Blanc, Rongxing Lu, Zonghua Zhang, and Ahmed Meddahi, editors, *Information and Communications Security - 20th International Conference, ICICS 2018, Lille, France, October 29-31, 2018, Proceedings*, volume 11149 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2018. doi: 10.1007/978-3-030-01950-1_11. URL `https://doi.org/10.1007/978-3-030-01950-1_11`.

[13] Alessandro Barenghi, Nicholas Mainardi, and Gerardo Pelosi. Systematic parsing of X.509: eradicating security issues with a parse tree. *J. Comput. Secur.*, 26(6):817–849, 2018. doi: 10.3233/JCS-171110. URL `https://doi.org/10.3233/JCS-171110`.

[14] Alessandro Barenghi, Nicholas Mainardi, and Gerardo Pelosi. A novel regular format for x.509 digital certificates. In Shahram Latifi, editor, *Information Technology - New Generations*, pages 133–139, Cham, 2018. Springer International Publishing. ISBN 978-3-319-54978-1. doi: 10.1007/978-3-319-54978-1_18. URL `https://doi.org/10.1007/978-3-319-54978-1_18`.

[15] Jean - Sé bastien Coron, Tancrè de Lepoint, and Mehdi Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In Hugo Krawczyk, editor, *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, volume 8383 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2014. doi: 10.1007/978-3-642-54631-0_18. URL `https://doi.org/10.1007/978-3-642-54631-0_18`.

[16] Emilia Kasper Ben Laurie, Adam Langley. Certificate Transparency. `https://www.certificate-transparency.org/`, 2014.

[17] DA Benson, M Cavanaugh, K Clark, I Karsch-Mizrachi, DJ Lipman, J Ostell, and EW Sayers. Genbank. nucleic acids res. Jan 2013. doi: 10.1093/nar/gks1195. The 9th International Symposium on String Processing and Information Retrieval.

[18] Gérard Berry and Ravi Sethi. From Regular Expressions to Deterministic Automata. *Theor. Comput. Sci.*, 48 (3):117–126, 1986. doi: 10.1016/0304-3975(86)90088-5.

[19] Hanno Böck. A Look at the PGP Ecosystem through the Key Server Data. IACR ePrint Archive: `http://eprint.iacr.org/2015/262`, 2015.

[20] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012. ISBN 978-3-642-32008-8. doi: 10.1007/978-3-642-32009-5. URL `https://doi.org/10.1007/978-3-642-32009-5`.

[21] Zvika Brakerski and Vinod Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. *SIAM J. Comput.*, 43(2):831–871, 2014. doi: 10.1137/120868669. URL `https://doi.org/10.1137/120868669`.

[22] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 309–325. ACM, 2012. ISBN 978-1-4503-1115-1. doi: 10.1145/2090236.2090262. URL `http://doi.acm.org/10.1145/2090236.2090262`.

[23] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association, 2017. URL `https://www.usenix.org/Conference/woot17/workshop-program/presentation/brasser`.

[24] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 114–129. IEEE Computer Society, 2014. doi: 10.1109/SP.2014.15. URL `https://doi.org/10.1109/SP.2014.15`.

[25] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In Kirda and Ristenpart [86], pages 1041–1056. URL `https://www.usenix.org/Conference/usenixsecurity17/technical-sessions/presentation/van-bulck`.

[26] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 991–1008. USENIX Association, 2018. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/bulck`.

[27] Jo Van Bulck, Danile Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[28] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994. URL `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf`.

[29] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880, 2007.

[30] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In Ray et al. [125], pages 668–679. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813700. URL `https://doi.org/10.1145/2810103.2813700`.

[31] Gizem S. Çetin, Yarkın Doröz, Berk Sunar, and William J. Martin. An Investigation of Complex Operations with Word-Size Homomorphic Encryption. *ePrint Archive*, (1195), 2015. `https://eprint.iacr.org/2015/1195.pdf`.

[32] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client ORAM. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL `https://www.ndss-symposium.org/ndss-paper/concuroram-high-throughput-stateless-parallel-multi-client-oram/`.

[33] Melissa Chase and Emily Shen. Substring-Searchable Symmetric Encryption. *PoPETs*, 2015(2): 263–281, 2015. URL `http://www.degruyter.com/view/j/popets.2015.2015.issue-2/popets-2015-0014/popets-2015-0014.xml`.

[34] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017. doi: 10.1007/978-3-319-70694-8_15. URL `https://doi.org/10.1007/978-3-319-70694-8_15`.

[35] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, 2016. ISBN 978-3-662-53886-9. doi: 10.1007/978-3-662-53887-6_1. URL `https://doi.org/10.1007/978-3-662-53887-6_1`.

[36] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[37] Noam Chomsky. Three Models for the Description of Language. *IRE Trans. Information Theory*, 2(3): 113–124, 1956. doi: 10.1109/TIT.1956.1056813.

[38] P.J. Cock, C.J. Fields, N. Goto, M.L. Heuer, and P.M. Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, 2010. doi: 10.1093/nar/gkp1137. URL `https://doi.org/10.1093/nar/gkp1137`.

[39] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and CRL. RFC 5280, 2008.

[40] David Coppit and Jiexin Lian. yagg: an easy-to-use generator for structured test inputs. In *ASE '05), Nov. 7-11, 2005, Long Beach, CA, USA*, pages 356–359. ACM, 2005. ISBN 1-58113-993-4.

[41] Intel Corporation. *Intel® Software Guard Extensions (Intel® SGX) SDK for Linux* OS*, v2.5 edition, May 2019. URL `https://download.01.org/intel-sgx/linux-2.5/docs/`.

[42] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[43] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In A. Juels, R. N. Wright, and S. De Capitani di Vimercati, editors, *Proc. of the 13th ACM Conf. on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 79–88. ACM, 2006. doi: 10.1145/1180405.1180417. URL `https://doi.org/10.1145/1180405.1180417`.

[44] Ivan Damgård and Mads Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In Kwangjo Kim, editor, *Public Key Cryptography, 4th Intl. Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proc.*, volume 1992 of *LNCS*, pages 119–136. Springer, 2001. ISBN 3-540-41658-7. doi: 10.1007/3-540-44586-2_9. URL `https://doi.org/10.1007/3-540-44586-2_9`.

[45] Quynh Dang, Stefan Santesson, Kathleen M. Moriarty, Daniel R. L. Brown, and Tim Polk. Internet X.509 Public Key Infrastructure: Additional Algorithms and Identifiers for DSA and ECDSA. RFC 5758, 2010.

[46] Antoine Delignat-Lavaud. BERserk Vulnerability Part 2: Certificate Forgery in Mozilla NSS. Tech. Rep.–Intel Advanced Threat Research Team. `http://www.intelsecurity.com/resources/wp-berserk-analysis-part-2.pdf`, 2014.

[47] Antoine Delignat-Lavaud. BERserk Vulnerability Part 1: RSA Signature Forgery Attack Due to Incorrect Parsing of ASN.1 Encoded DigestInfo in PKCS1 v1.5. Tech. Rep.–Intel Advanced Threat Research Team. `http://www.intelsecurity.com/resources/wp-berserk-analysis-part-1.pdf`, 2014.

[48] Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Efficient and private access to outsourced data. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 710–719. IEEE Computer Society, 2011. doi: 10.1109/ICDCS.2011.37. URL `https://doi.org/10.1109/ICDCS.2011.37`.

[49] Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Supporting concurrency and multiple indexes in private access to outsourced data. *J. Comput. Secur.*, 21(3): 425–461, 2013. doi: 10.3233/JCS-130468. URL `https://doi.org/10.3233/JCS-130468`.

[50] Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Shuffle index: Efficient and private access to outsourced data. *ACM Trans. Storage*, 11(4):19:1–19:55, 2015. doi: 10.1145/2747878. URL `https://doi.org/10.1145/2747878`.

[51] Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Access privacy in the cloud. In Pierangela Samarati, Indrajit Ray, and Indrakshi Ray, editors, *From Database to Cyber Security - Essays Dedicated to Sushil Jajodia on the Occasion of His 70th Birthday*, volume 11170 of *Lecture Notes in Computer Science*, pages 186–205. Springer, 2018. doi: 10.1007/978-3-030-04834-1_10. URL `https://doi.org/10.1007/978-3-030-04834-1_10`.

[52] Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Enforcing authorizations while protecting access confidentiality. *J. Comput. Secur.*, 26(2):143–175, 2018. doi: 10.3233/JCS-171004. URL https://doi.org/10.3233/JCS-171004.

[53] Sabrina De Capitani di Vimercati, Sara Foresti, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Three-server swapping for access confidentiality. *IEEE Trans. Cloud Comput.*, 6(2):492–505, 2018. doi: 10.1109/TCC.2015.2449993. URL https://doi.org/10.1109/TCC.2015.2449993.

[54] Yarkin Doröz and Berk Sunar. Flattening NTRU for evaluation key free homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2016:315, 2016. URL http://eprint.iacr.org/2016/315.

[55] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012. URL http://eprint.iacr.org/2012/144.

[56] Sebastian Faust, Carmit Hazay, and Daniele Venturi. Outsourced pattern matching. *Int. J. Inf. Sec.*, 17(3):327–346, 2018. doi: 10.1007/s10207-017-0374-0. URL https://doi.org/10.1007/s10207-017-0374-0.

[57] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi: 10.1145/1082036.1082039. URL https://doi.org/10.1145/1082036.1082039.

[58] Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Trans. Algorithms*, 7(1): 10:1–10:21, 2010. doi: 10.1145/1868237.1868248. URL https://doi.org/10.1145/1868237.1868248.

[59] Paul Flicek et. al. Ensembl Genome Browser, 2000. www.ensembl.org/.

[60] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009. doi: 10.1145/1536414.1536440. URL http://doi.acm.org/10.1145/1536414.1536440.

[61] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully Homomorphic Encryption with Polylog Overhead. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. ISBN 978-3-642-29010-7. doi: 10.1007/978-3-642-29011-4. URL https://doi.org/10.1007/978-3-642-29011-4.

[62] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013. ISBN 978-3-642-39076-0. doi: 10.1007/978-3-642-39077-7_1. URL https://doi.org/10.1007/978-3-642-39077-7_1.

[63] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013. ISBN 978-3-642-40040-7. doi: 10.1007/978-3-642-40041-4. URL https://doi.org/10.1007/978-3-642-40041-4.

[64] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194. ACM, 1987. doi: 10.1145/28395.28416. URL https://doi.org/10.1145/28395.28416.

[65] Google Inc. BoringSSL. https://boringssl.googlesource.com/boringssl/, 2016.

[66] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2012. http://gmplib.org/.

[67] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking Web Applications Built On Top of Encrypted Data. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proc. of the 2016 ACM SIGSAC Conf. on Computer*

and Communications Security, Vienna, Austria, October 24-28, 2016, pages 1353–1364. ACM, 2016. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978351. URL `https://doi.org/10.1145/2976749.2978351`.

[68] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In Kirda and Ristenpart [86], pages 217–233. URL `https://www.usenix.org/Conference/usenixsecurity17/technical-sessions/presentation/gruss`.

[69] Florian Hahn, Nicolas Loza, and Florian Kerschbaum. Practical and Secure Substring Search. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proc. of the 2018 Intl. Conf. on Management of Data, SIGMOD Conf. 2018, Houston, TX, USA, June 10-15, 2018*, pages 163–176. ACM, 2018. doi: 10.1145/3183713.3183754. URL `https://doi.org/10.1145/3183713.3183754`.

[70] Michael. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978. ISBN 0201029553.

[71] Philip Hazel. PCRE - Perl Compatible Regular Expressions, 2015. `https://www.pcre.org`.

[72] Russell Housley, Burt Kaliski, and Jim Schaad. Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL). RFC 4055, 2005.

[73] IETF OpenPGP WorkGroup. Open Specification for Pretty Good Privacy. `https://datatracker.ietf.org/wg/openpgp/charter/`, 2017.

[74] Intel Corporation. Description and mitigation overview for l1 terminal fault, 2018. URL `https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault`.

[75] Intel Corporation. Deep dive: Special register buffer data sampling, 2020. URL `https://software.intel.com/security-software-guidance/insights/deep-dive-special-register-buffer-data-sampling`.

[76] Intel Corporation. Deep dive: Load value injection, 2020. URL `https://software.intel.com/security-software-guidance/insights/deep-dive-load-value-injection`.

[77] International Telecommunication Union. Recommendation ITU-T X.509: Open Systems Interconnection - Public-key and Attribute Certificate Frameworks. `https://www.itu.int/rec/T-REC-X.509-201210-I`, 2012.

[78] International Telecommunication Union. Recommendation ITU-T X.690: Information technology - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). `https://www.itu.int/rec/T-REC-X.690`, 2015.

[79] International Telecommunication Union. Data networks, Open System Comm.s and Security Recommendations. `https://www.itu.int/rec/T-REC-X`, 2016.

[80] Yu Ishimaki, Hiroki Imabayashi, and Hayato Yamana. Private Substring Search on Homomorphically Encrypted Data. In *2017 IEEE Intl. Conf. on Smart Computing, SMARTCOMP 2017, Hong Kong, China, May 29-31, 2017*, pages 1–6. IEEE Computer Society, 2017. ISBN 978-1-5090-6517-2. doi: 10.1109/SMARTCOMP.2017.7947038. URL `https://doi.org/10.1109/SMARTCOMP.2017.7947038`.

[81] Dan Kaminsky, Meredith L. Patterson, and Len Sassaman. PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure. In Radu Sion, editor, *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers*, volume 6052 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2010. ISBN 978-3-642-14576-6. doi: 10.1007/978-3-642-14577-3_22.

[82] Juha Kärkkäinen and Peter Sanders. Simple Linear Work Suffix Array Construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th Intl. Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proc.*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003. ISBN 3-540-40493-7. doi: 10.1007/3-540-45061-0_73. URL `https://doi.org/10.1007/3-540-45061-0_73`.

[83] Alan H. Karp and Horace P. Flatt. Measuring Parallel Processor Performance. *Commun. ACM*, 33(5):539–543, 1990. doi: 10.1145/78607.78614. URL `https://doi.org/10.1145/78607.78614`.

[84] Alhassan Khedr, P. Glenn Gulak, and Vinod Vaikuntanathan. SHIELD: scalable homomorphic implementation of encrypted data-classifiers. *IEEE Trans. Computers*, 65(9):2848–2858, 2016. doi: 10.1109/TC.2015.2500576. URL `https://doi.org/10.1109/TC.2015.2500576`.

[85] Aviad Kipnis and Eliphaz Hibshoosh. Efficient Methods for Practical Fully Homomorphic Symmetric-key Encrypton, Randomization and Verification. `http://eprint.iacr.org/2012/637`.

[86] Engin Kirda and Thomas Ristenpart, editors. *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, 2017. USENIX Association. URL `https://www.usenix.org/conference/usenixsecurity17`.

[87] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *Machine Learning: ECML 2004, 15th European Conference on Machine Learning, Pisa, Italy, September 20-24, 2004, Proceedings*, volume 3201 of *Lecture Notes in Computer Science*, pages 217–226. Springer, 2004. doi: 10.1007/978-3-540-30115-8_22. URL `https://doi.org/10.1007/978-3-540-30115-8_22`.

[88] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997. ISBN 0201896842 9780201896848.

[89] Werner Koch and David Shaw. The GNU Privacy Guard. `https://www.gnupg.org`, 1997.

[90] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[91] Benjamin Leiding and Andreas Dähn. Dead Letters to Alice - Reachability of E-Mail Addresses in the PGP Web of Trust. *CoRR*, abs/1605.03162, 2016.

[92] Iraklis Leontiadis and Ming Li. Storage Efficient Substring Searchable Symmetric Encryption. In Aziz Mohaisen and Qian Wang, editors, *Proc. of the 6th Intl. Workshop on Security in Cloud Computing, SCC@AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 3–13. ACM, 2018. doi: 10.1145/3201595.3201598. URL `https://doi.org/10.1145/3201595.3201598`.

[93] Serguei Leontiev and Dennis Shefanovski. Using the GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and CRL Profile. RFC 4491, 2006.

[94] Jing Li and Licheng Wang. Noise-Free Symmetric Fully Homomorphic Encryption Based on Non-Commutative Rings. IACR ePrint Archive, Report 2015/641, 2015. `https://eprint.iacr.org/2015/641`.

[95] John Linn. Privacy Enhancement for Internet Electronic Mail: Message Encryption and Authentication. RFC 1421, 1993.

[96] Helger Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. In J. Zhou, J. López, R. H. Deng, and F. Bao, editors, *Information Security, 8th Intl. Conf., ISC 2005, Singapore, September 20-23, 2005, Proc.*, volume 3650 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2005. ISBN 3-540-29001-X. doi: 10.1007/11556992_23. URL `https://doi.org/10.1007/11556992_23`.

[97] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[98] Dongxi Liu. Practical Fully Homomorphic Encryption without Noise Reduction. `http://eprint.iacr.org/2015/468`.

[99] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In Howard J. Karloff and Toniann Pitassi, editors, *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1219–1234. ACM, 2012. doi: 10.1145/2213977.2214086. URL `https://doi.org/10.1145/2213977.2214086`.

[100] Fucai Luo, Fuqun Wang, Kunpeng Wang, Jie Li, and Kefei Chen. Lwr-based fully homomorphic encryption, revisited. *Security and Communication Networks*, 2018:5967635:1–5967635:12, 2018. doi: 10.1155/2018/5967635. URL `https://doi.org/10.1155/2018/5967635`.

[101] Fucai Luo, Fuqun Wang, Kunpeng Wang, and Kefei Chen. Fully homomorphic encryption based on the ring learning with rounding problem. *IET Inf. Secur.*, 13(6):639–648, 2019. doi: 10.1049/iet-ifs.2018.5427. URL `https://doi.org/10.1049/iet-ifs.2018.5427`.

[102] Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. Privacy preserving substring search protocol with polylogarithmic communication cost. In David Balenson, editor, *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, pages 297–312. ACM, 2019. doi: 10.1145/3359789.3359842. URL `https://doi.org/10.1145/3359789.3359842`.

[103] Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. Plaintext recovery attacks against linearly decryptable fully homomorphic encryption schemes. *Comput. Secur.*, 87, 2019. doi: 10.1016/j.cose.2019.101587. URL `https://doi.org/10.1016/j.cose.2019.101587`.

[104] Nicholas Mainardi, Davide Sampietro, Alessandro Barenghi, and Gerardo Pelosi. Efficient oblivious substring search via architectural support. In Kevin Butler, editor, *Proceedings of the 36th Annual Computer Security Applications Conference, ACSAC 2020, Austin, TX, USA, December 07-11, 2020*. ACM, 2020. doi: 10.1145/3427228.3427296. URL `https://doi.org/10.1145/3427228.3427296`.

[105] Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi: 10.1137/0222058. URL `https://doi.org/10.1137/0222058`.

[106] Moxie Marlinspike. Internet Explorer SSL Vulnerability. `https://moxie.org/ie-ssl-chain.txt`, 2002.

[107] Moxie Marlinspike. Null Prefix Attacks against SSL/TLS Certificates. `https://moxie.org/papers/null-prefix-attacks.pdf`, 2009.

[108] Paulo Martins, Leonel Sousa, and Artur Mariano. A survey on fully homomorphic encryption: An engineering perspective. *ACM Comput. Surv.*, 50(6):83:1–83:33, 2018. doi: 10.1145/3124441. URL `https://doi.org/10.1145/3124441`.

[109] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. *PoPETs*, 2016(2), 2016. doi: 10.1515/popets-2016-0010. URL `https://doi.org/10.1515/popets-2016-0010`.

[110] Y. Minsky, J. Clizbe, and K. Fiskerstrand. Synchronizing Key Server (SKS) Software Package. `https://bitbucket.org/skskeyserver/sks-keyserver/wiki/Home`, 2015.

[111] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 279–296. IEEE Computer Society, 2018. ISBN 978-1-5386-4353-2. doi: 10.1109/SP.2018.00045. URL `https://doi.org/10.1109/SP.2018.00045`.

[112] Tarik Moataz and Erik-Oliver Blass. Oblivious Substring Search with Updates. *IACR Cryptology ePrint Archive*, 2015:722, 2015. URL `http://eprint.iacr.org/2015/722`.

[113] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 735–763. Springer, 2016. doi: 10.1007/978-3-662-49896-5_26. URL `https://doi.org/10.1007/978-3-662-49896-5_26`.

[114] Harika Narumanchi, Dishant Goyal, Nitesh Emmadi, and Praveen Gauravaram. Performance Analysis of Sorting of FHE Data: Integer-Wise Comparison vs Bit-Wise Comparison. In Leonard Barolli, Makoto Takizawa, Tomoya Enokido, Hui-Huang Hsu, and Chi-Yi Lin, editors, *31st IEEE International Conference on Advanced Information Networking and Applications, AINA 2017, Taipei, Taiwan, March 27-29, 2017*, pages 902–908. IEEE Computer Society, 2017. doi: 10.1109/AINA.2017.85. URL `https://doi.org/10.1109/AINA.2017.85`.

[115] Koji Nuida. A Simple Framework for Noise-Free Construction of Fully Homomorphic Encryption from a Special Class of Non-Commutative Groups. IACR ePrint Archive, 2014. `http://eprint.iacr.org/2014/097`.

[116] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 227–240. USENIX Association, 2018. URL `https://www.usenix.org/conference/atc18/presentation/oleksenko`.

[117] OpenSSL Foundation. OpenSSL: Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/`, 2016.

[118] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, Intl. Conf. on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999. ISBN 3-540-65889-0. doi: 10.1007/3-540-48910-X_16. URL `https://doi.org/10.1007/3-540-48910-X_16`.

[119] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 425–436. ACM, 2011. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993548.

[120] Terence J. Parr and Russell W. Quong. Adding Semantic and Syntactic Predicates To LL(K): Pred-LL(K). In *Proceedings of the 5th International Conference on Compiler Construction*, CC '94, pages 263–277, London, UK, UK, 1994. Springer-Verlag. ISBN 3-540-57877-3.

[121] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 871–882. IEEE Computer Society, 2018. doi: 10.1109/FOCS.2018.00087. URL `https://doi.org/10.1109/FOCS.2018.00087`.

[122] Tim Polk, Russell Housley, and Larry Bassham. Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3279, 2002.

[123] Shiyue Qin, Fucai Zhou, Zongye Zhang, and Zifeng Xu. Privacy-preserving substring search on multi-source encrypted gene data. *IEEE Access*, 8:50472–50484, 2020. doi: 10.1109/ACCESS.2020.2980375. URL `https://doi.org/10.1109/ACCESS.2020.2980375`.

[124] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *42th IEEE Symposium on Security and Privacy (S&P'21)*, 2021. URL `https://download.vusec.net/papers/crosstalk_sp21.pdf`. Intel Bounty Reward.

[125] Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, 2015. ACM. ISBN 978-1-4503-3832-5.

[126] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009. doi: 10.1145/1568318.1568324. URL `http://doi.acm.org/10.1145/1568318.1568324`.

[127] Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity verification for path oblivious-ram. In *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*, pages 1–6. IEEE, 2013. ISBN 978-1-4799-1365-7. doi: 10.1109/HPEC.2013.6670339. URL `https://doi.org/10.1109/HPEC.2013.6670339`.

[128] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 415–430. USENIX Association, 2015. URL `https://www.usenix.org/Conference/usenixsecurity15/technical-sessions/presentation/ren-ling`.

[129] R L Rivest, L Adleman, and M L Dertouzos. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[130] Cédric Van Rompay, Refik Molva, and Melek Önen. A Leakage-Abuse Attack Against Multi-User Searchable Encryption. *PoPETs*, 2017(3):168, 2017. doi: 10.1515/popets-2017-0034. URL `https://doi.org/10.1515/popets-2017-0034`.

[131] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 198–217. IEEE Computer Society, 2016. doi: 10.1109/SP.2016.20. URL `https://doi.org/10.1109/SP.2016.20`.

[132] Davide Sampietro. ObSQRE : efficient full-text index for oblivious substring search queries with Intel SGX. Master's thesis, Politecnico di Milano, 2019. URL `http://hdl.handle.net/10589/150515`.

[133] Davide Sampietro and Nicholas Mainardi. ObSQRE: Oblivious Substring Queries on Remote Enclave, 2020. URL `https://github.com/DavideSampietro/ObSQRE`.

[134] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. Security Applications of Formal Language Theory. *IEEE Systems Journal*, 7(3):489–500, 2013.

[135] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. URL `http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_02B-4_Sasy_paper.pdf`.

[136] Richard Donald Schafer. *An introduction to nonassociative algebras*. Dover Publications, Inc., Mineola, New York, 2017.

[137] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with intel SGX. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, volume 11543 of *Lecture Notes in Computer Science*, pages 177–196. Springer, 2019. doi: 10.1007/978-3-030-22038-9_9. URL `https://doi.org/10.1007/978-3-030-22038-9_9`.

[138] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. URL `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/`.

[139] Kana Shimizu, Koji Nuida, and Gunnar Rätsch. Efficient privacy-preserving string search and an application in genomics. *Bioinformatics*, 32(11), 2016. doi: 10.1093/bioinformatics/btw050. URL `https://doi.org/10.1093/bioinformatics/btw050`.

[140] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 317–328. ACM, 2016. ISBN 978-1-4503-4233-9. doi: 10.1145/2897845.2897885. URL `https://doi.org/10.1145/2897845.2897885`.

[141] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptography*, 71(1):57–81, 2014. doi: 10.1007/s10623-012-9720-4. URL `https://doi.org/10.1007/s10623-012-9720-4`.

[142] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310. ACM, 2013. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516660. URL `https://doi.org/10.1145/2508859.2516660`.

[143] Larry Stefonic and Todd Ouska. The wolfSSL Embedded TLS Library, 2019. URL `https://www.wolfssl.com/`.

[144] Damien Stehlé and Ron Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 27–47. Springer, 2011. doi: 10.1007/978-3-642-20465-4_4. URL `https://doi.org/10.1007/978-3-642-20465-4_4`.

[145] Stephen C Johnson. Yacc: Yet Another Compiler-Compiler. `http://dinosaur.compilertools.net/yacc/index.html`.

[146] Julien P. Stern. A new efficient all-or-nothing disclosure of secrets protocol. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology - ASIACRYPT '98, International Conference on the Theory and Applications of Cryptology and Information Security, Beijing, China, October 18-22, 1998, Proceedings*, volume 1514 of *LNCS*, pages 357–371. Springer, 1998. doi: 10.1007/3-540-49649-1_28. URL `https://doi.org/10.1007/3-540-49649-1_28`.

[147] Mikhail Strizhov, Zachary Osman, and Indrajit Ray. Substring Position Search over Encrypted Cloud Data Supporting Efficient Multi-User Setup. *Future Internet*, 8(3), 2016. doi: 10.3390/fi8030028. URL `https://doi.org/10.3390/fi8030028`.

[148] Symantec Corp. Symantec Encryption (PGP) Docs. Article Tech202483. `https://knowledge.broadcom.com/external/article?legacyId=TECH202483`, 2016.

[149] Cong Tian, Chu Chen, Zhenhua Duan, and Liang Zhao. Differential testing of certificate validation in ssl/tls implementations: An rfc-guided approach. 28(4), October 2019. ISSN 1049-331X. doi: 10.1145/3355048. URL `https://doi.org/10.1145/3355048`.

[150] Marc Tiehuis. libhcs: A partially Homomorphic C library, 2015. `https://github.com/tiehuis/libhcs/tree/master/include/libhcs`.

[151] Boaz Tsaban and Noam Lifshitz. Cryptanalysis of the MORE symmetric key fully homomorphic encryption scheme. *J. Mathematical Cryptology*, 9(2):75–78, 2015. doi: 10.1515/jmc-2014-0013. URL `https://doi.org/10.1515/jmc-2014-0013`.

[152] Sean Turner, Daniel R. L. Brown, Kelvin Yiu, Russell Housley, and Tim Polk. Elliptic Curve Cryptography Subject Public Key Information. RFC 5480, 2009.

[153] The UniProt Consortium. UniProt: the universal protein knowledgebase. *Nucleic Acids Research*, 46(5): 2699–2699, 02 2018. ISSN 0305-1048. doi: 10.1093/nar/gky092. URL `https://doi.org/10.1093/nar/gky092`.

[154] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010. doi: 10.1007/978-3-642-13190-5_2. URL `https://doi.org/10.1007/978-3-642-13190-5_2`.

[155] Bing Wang, Wei Song, Wenjing Lou, and Y. Thomas Hou. Privacy-preserving pattern matching over encrypted genetic data in cloud computing. In *2017 IEEE Conf. on Computer Communications, INFOCOM 2017, Atlanta, GA, USA, May 1-4, 2017*, pages 1–9. IEEE, 2017. ISBN 978-1-5090-5336-0. doi: 10.1109/INFOCOM.2017.8057178. URL `https://doi.org/10.1109/INFOCOM.2017.8057178`.

[156] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In Ray et al. [125], pages 850–861. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813634. URL `https://doi.org/10.1145/2810103.2813634`.

[157] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 215–226. ACM, 2014. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660314. URL `https://doi.org/10.1145/2660267.2660314`.

[158] Yongge Wang and Qutaibah M. Malluhi. Privacy Preserving Computation in Cloud Using Noise-Free Fully Homomorphic Encryption (FHE) Schemes. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, volume 9878 of *Lecture Notes in Computer Science*, pages 301–323. Springer, 2016. ISBN 978-3-319-45743-7. doi: 10.1007/978-3-319-45744-4. URL https://doi.org/10.1007/978-3-319-45744-4.

[159] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. Sgxjail: Defeating enclave malware via confinement. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pages 353–366. USENIX Association, 2019. URL https://www.usenix.org/conference/raid2019/presentation/weiser.

[160] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: a parallel oblivious file system. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 977–988. ACM, 2012. doi: 10.1145/2382196.2382299. URL https://doi.org/10.1145/2382196.2382299.

[161] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.45. URL https://doi.org/10.1109/SP.2015.45.

APPENDIX $\mathcal{A}$

Appendix

1 Make OctoM Multiplicatively Homomorphic

In this section, we show how to modify the homomorphic multiplication MUL of the `OctoM` scheme in order to satisfy the evaluation correctness property: that is, given two ciphertexts $C_1, C_2 \in \mathbb{Z}_N^{8\times 8}$ and the ciphertext $C_{mul} = \textsc{Mul}(evk, C_1, C_2) \in \mathbb{Z}_N^{8\times 8}$, then $\textsc{Dec}(k, C_{mul}) = \textsc{Dec}(k, C_1) \times \textsc{Dec}(k, C_2) \bmod N$. In particular, first we show why the MUL operation, as presented in [158], does not satisfy the evaluation correctness property; then, we discuss the additional constraints in the key generation algorithm which make `OctoM` multiplicatively homomorphic, and thus fully homomorphic.

$$\begin{aligned}
\textsc{Dec}(k, C_{mul}) &= \phi^{-1}(\mathbb{1} \cdot M \cdot C_{mul} \cdot M^{-1}) \cdot v^T = \phi^{-1}(\mathbb{1} \cdot M \cdot M^{-1} \cdot A^l_{m'_2} \cdot A^l_{m'_1} \cdot A^l_{-1} \cdot M \cdot M^{-1}) \cdot v^T \\
&= \phi^{-1}(\mathbb{1} \cdot A^l_{m'_2} \cdot A^l_{m'_1} \cdot A^l_{-1}) \cdot v^T = \phi^{-1}(m'_2 \cdot A^l_{m'_1} \cdot A^l_{-1}) \cdot v^T \\
&= \phi^{-1}((m'_1 * m'_2) \cdot A^l_{-1}) \cdot v^T = \phi^{-1}(\phi(-i + z_{-1}) * (m'_1 * m'_2)) \cdot v^T \\
&= \phi^{-1}(\phi(-i + z_{-1})) * \phi^{-1}(m'_1 * m'_2) \cdot v^T \\
&= (-i + z_{-1}) * (\phi^{-1}(\phi(m_1 i + z_1) * \phi(m_2 i + z_2))) \cdot v^T \\
&= (-i + z_{-1}) * (\phi^{-1}(\phi((m_1 i + z_1) * (m_2 i + z_2)))) \cdot v^T \\
&= (-i + z_{-1}) * ((m_1 i + z_1) * (m_2 i + z_2)) \cdot v^T \\
&= \boxed{-i((m_1 i)(m_2 i))} = -i(-m_1 m_2) = m_1 m_2 i
\end{aligned} \tag{A.1}$$

According to [158], given two ciphertexts $C_1, C_2 \in \mathbb{Z}_N^{8\times 8}$ and their homomorphic product $C_{mul} = \textsc{Mul}(evk, C_1, C_2)$, the decryption algorithm $\textsc{Dec}(k, C_{mul}) = \phi^{-1}(\mathbb{1} \cdot M \cdot C_{mul} \cdot M^{-1}) \bmod V$ retrieves the plaintext value following the calculations shown in Eq. (A.1). In particular, the erroneous derivation in this chain of equations is shadowed in gray in Eq. (A.1). Denoting as a_{mul} the octonion computed by $\phi^{-1}(\mathbb{1}{\cdot}M{\cdot}C_{mul}{\cdot}M^{-1})$, we can write it as $a_{mul} = m_1 m_2 i + z_{mul}$; the issue with the erroneous derivation in Eq. (A.1) is that the octonion z_{mul} is not necessarily in the isotropic subspace V, which means that the modulo V operation performed as the inner product $a_{mul} \cdot v^T$ may not yield $m_1 m_2 i$.

The fact that z_{mul} does not necessarily belong to V is proven as follows. Eq. (A.2) shows the multiplication of the three octonions in the last-but-one line of Equation A.1:

$$
\begin{aligned}
((-i + z_{-1}) * ((m_1 i + z_1) * (m_2 i + z_2))) \cdot v^T = {} & ((-i + z_{-1}) * (-m_1 m_2 + m_1(i * z_2) + \\
& (z_1 * i) m_2 + z_1 * z_2)) \cdot v^T \\
= {} & (m_1 m_2 i - m_1(i * (i * z_2)) - m_2(i * (z_1 * i)) - i * (z_1 * z_2) - m_1 m_2 z_{-1} \\
+ {} & m_1(z_{-1} * (i * z_2)) + m_2(z_{-1} * (z_1 * i)) + z_{-1} * (z_1 * z_2)) \cdot v^T \\
= {} & (m_1 m_2 i + z_{mul}) \cdot v^T
\end{aligned}
\tag{A.2}
$$

Eq. (A.2) points out that z_{mul} is equal to the sum of several octonion terms and some of them (the ones highlighted by a light gray background in Eq. (A.2)) are not necessarily octonions belonging to V, making z_{mul} not necessarily belonging to V too. Indeed, they contain the imaginary unit $i \notin V$, thus their sum z_{mul} may be out of V. Obviously, the presence of these additional terms makes the decryption erroneous, since these terms are added to the term $m_1 m_2 i$ and are not canceled out by the inner product with the vector $v \in V^{\perp}$. We practically observed decryption failures due to this issue in our pilot implementation.

Modifications to `OctoM` Homomorphic Multiplication

We now describe how to compensate for the additional terms that are not in the totally isotropic subspace V, employing a monodimensional V space. We remark that such a choice does not affect the security of the cryptosystem, as reported in [158]. In case of a monodimensional subspace, all the vectors in V are obtained as rz, where r is a random integer in $\mathbb{Z}_N$ and z is the isotropic octonion generating the subspace V. Moreover, for a generic octonion a, it holds that $a^2 = 2\Re(a)a - \|a\|^2\mathbb{1}$ (Thm. 2 in [158]), therefore for an isotropic octonion z, $z^2 = 2\Re(z)z - \|z\|^2\mathbb{1} = 2\Re(z)z$. This property allows to show that every monodimensional totally isotropic subspace V is closed under octonion multiplication, since, for every two generic octonions in V, namely $z_1 = rz$, $z_2 = sz$, their product is still in V, as $z_1 * z_2 = rsz^2 = 2rs\Re(z)z$. Henceforth, we denote by $z = [z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8]$ the isotropic octonion generating the totally isotropic subspace V.

In order to make `OctoM` multiplicatively homomorphic, we need to somehow get rid of the additional octonions highlighted in Eq. (A.2). The main idea is to append a vector v, orthogonal to all the additional octonions, to the secret key computed by the KEYGEN algorithm. However, this solution is viable only if the additional octonions added to the result are of the same form regardless of the number of homomorphic multiplications performed to compute the ciphertext to be decrypted.

Since all these additional terms involve a multiplication of an octonion with the first imaginary unit i, we analyze the octonion products $i * z$ and $z * i$. The following equalities show the computation of such products:

$$\begin{aligned} i * z &= [-z_2, z_1, -z_4, z_3, -z_6, z_5, z_8, -z_7] \\ z * i &= [-z_2, z_1, z_4, -z_3, z_6, -z_5, -z_8, z_7] \end{aligned} \tag{A.3}$$

The product of an octonion with the imaginary unit is not commutative. However, octonion algebra is *alternative*, therefore Artin's theorem [136] can be applied:

Theorem A.1 (Artin's Theorem [136])**.** *An algebra $\mathbb{A}$ is alternative if and only if, for any two elements a, b in the support of the algebra, the three following equalities hold:*

$$\begin{aligned} a * (a * b) &= (a * a) * b \\ (a * b) * a &= a * (b * a) \\ (b * a) * a &= b * (a * a) \end{aligned}$$

Artin's theorem equivalently states that the sub-algebra generated by two elements of an alternative algebra is associative. By applying this theorem on the two octonions i, z, we can derive the following equalities, related to the additional terms appearing in Eq. (A.2):

$$\begin{aligned} & i * (z * i) = (i * z) * i \\ & (z * i) * z = z * (i * z) \\ & z * (z * i) = (z * z) * i = 2\Re(z)(z * i) \\ & (i * z) * z = i * (z * z) = 2\Re(z)(i * z) \\ & (z * i) * i = z * (i * i) = -z \end{aligned} \tag{A.4}$$

These equalities will be employed to verify the following statement:

Statement A.1. *Consider $h > 0$ octonions $a_j \in \mathbb{O}(\mathbb{Z}_N^8)$, $1 \leq j \leq h$, of the form $a_j = m_j i + z_j = m_j i + r_j z$, where $m_j, r_j \in \mathbb{Z}_N$ and $z \in \mathbb{O}(\mathbb{Z}_N^8)$ is an isotropic octonion. For every $h > 0$, the product of these octonions is:*

$$\begin{aligned} a_1 * a_2 * \cdots * a_h = i^h \prod_{j=1}^{h} m_j &+ R_z z + R_{iz}(i * z) \\ &+ R_{zi}(z * i) + R_{izi}(i * z * i) \\ &+ R_{ziz}(z * i * z) \end{aligned} \tag{A.5}$$

where R_z, R_{iz}, R_{zi}, R_{izi}, R_{ziz} are integers in $\mathbb{Z}_N$.

Before proving Statement A.1, we present some additional equalities involving the imaginary unit i and the isotropic octonion z. These equalities, which will be employed in the proof, are shown in Eq. (A.6). All these relationships hinge upon the associative multiplication between i and z. In addition, in the last step of the bottom relationships, we exploit the fact that $i * z$ and $z * i$ are isotropic octonions: indeed, since octonions are a normed algebra, $\|a * b\| = \|a\| \cdot \|b\|$, thus $\|i * z\| = \|i\| \cdot \|z\| = 1 \cdot 0 = 0$, and similarly for the octonion $z * i$. Therefore, since for an isotropic octonion z, $z^2 = 2\Re(z)z$, then $(i * z)^2 = 2\Re(i * z)(i * z)$ and similarly for the octonion $z * i$.

We now prove Statement A.1.

$$\begin{aligned}
(i*(z*i))*i &= ((i*z)*i)*i = (i*z)*(i*i) = -(i*z) \\
(z*(i*z))*z &= ((z*i)*z)*z = (z*i)*(z*z) = 2\Re(z)((z*i)*z) \\
(z*(i*z))*i &= (z*i)*(z*i) = (z*i)^2 = 2\Re(z*i)(z*i) \\
(i*(z*i))*z &= (i*z)*(i*z) = (i*z)^2 = 2\Re(i*z)(i*z)
\end{aligned} \tag{A.6}$$

$$\begin{aligned}
a_{mul}*a_{h+1} &= \left(i^h \prod_{j=1}^{h} m_j + R_z z + R_{iz}(i*z) + R_{zi}(z*i) + R_{izi}(i*z*i) + R_{ziz}(z*i*z) \right) * (m_{h+1}i + r_{h+1}z) \\
&= i^{h+1}\prod_{j=1}^{h+1} m_j + \left(\prod_{j=1}^{h} m_j\right) r_{h+1}(i^h*z) + R_z m_{h+1}(z*i) + R_z r_{h+1}(z*z) + R_{iz}m_{h+1}((i*z)*i) \\
&\quad +R_{iz}r_{h+1}((i*z)*z) \ +R_{zi}m_{h+1}((z*i)*i) + R_{zi}r_{h+1}((z*i)*z) +R_{izi}m_{h+1}((i*z*i)*i) \\
&\quad + R_{izi}r_{h+1}((i*z*i)*z) + R_{ziz}m_{h+1}((z*i*z)*i) + R_{ziz}r_{h+1}((z*i*z)*z) \\
&= i^{h+1}\prod_{j=1}^{h+1} m_j + \left(\prod_{j=1}^{h} m_j\right) r_{h+1}(i^h*z) + R_z m_{h+1}(z*i) + R_z r_{h+1}2\Re(z)z + R_{iz}m_{h+1}((i*z)*i) \\
&\quad +R_{iz}r_{h+1}2\Re(z)(i*z) \ -R_{zi}m_{h+1}z + R_{zi}r_{h+1}((z*i)*z) -R_{izi}m_{h+1}(i*z) \\
&\quad + R_{izi}r_{h+1}2\Re(i*z)(i*z) + R_{ziz}m_{h+1}2\Re(z*i)(z*i) + R_{ziz}r_{h+1}2\Re(z)(z*i*z) \\
&= i^{h+1}\prod_{j=1}^{h+1} m_j + R'_z z + R'_{iz}(i*z) + R'_{izi}(i*(z*i)) + R'_{ziz}(z*(i*z)) + R'_{zi}(z*i)
\end{aligned} \tag{A.7}$$

Proof. The statement trivially holds for $h = 1$. Indeed, $a_1 = m_1 i + r_1 z$, which is equivalent to Eq. (A.5) where $R_z = r_1$ and $R_{zi} = R_{zi} = R_{izi} = R_{ziz} = 0$. Next, we consider $h = 2$ octonions $a_1 = m_2 i + r_2 z$, $a_2 = m_2 i + r_2 z$. If we multiply these octonions, we obtain:

$$\begin{aligned}
a_1 * a_2 &= (m_1 i + r_1 z) * (m_2 i + r_2 z) \\
&= -m_1 m_2 + m_1 r_2 (i*z) + r_1 m_2 (z*i) + r_1 r_2 z^2 \\
&= -m_1 m_2 + m_1 r_2 (i*z) + r_1 m_2 (z*i) + r_1 r_2 2\Re(z) z
\end{aligned}$$

Since $a_1 * a_2$ is equivalent to Eq. (A.5) where $R_z = r_1 r_2 2\Re(z)$, $R_{iz} = m_1 r_2$, $R_{zi} = m_2 r_1$ and $R_{izi} = R_{ziz} = 0$, Statement A.1 holds for $h = 2$ too. We are now ready for the induction step. Assume that the statement holds for an octonion a_{mul} obtained by multiplying h octonions a_j, $j = 1, \ldots, h$. Eq. (A.7) shows that, if we multiply another octonion $a_{h+1} = m_{h+1} i + r_{h+1} z$, the statement holds too. Here, we highlighted the derivations where we employ relationships from either Eq. (A.4) or Eq. (A.6). Lastly, we remark that the octonion $\left(\prod_{j=1}^{h} m_j\right) r_{h+1}(i^h * z)$, which is part of the computation shown in Eq. (A.7), in the last step is either grouped in the term $R'_z z$ or in the term $R'_{iz}(i*z)$, depending on the parity of h: indeed, if h is even, then $i^h = \pm 1$ is a real number, and so $\left(\prod_{j=1}^{h} m_j\right) r_{h+1}(i^h * z) = \pm \left(\prod_{j=1}^{h} m_j\right) r_{h+1} z$; instead, if h is odd, then $i^h = \pm i$ is an imaginary number, and so $\left(\prod_{j=1}^{h} m_j\right) r_{h+1}(i^h * z) = \pm \left(\prod_{j=1}^{h} m_j\right) r_{h+1}(i * z)$. □

We now show how to apply Statement A.1 to the homomorphic multiplication of

`OctoM`. Given h ciphertexts $C_1, C_2, \dots C_h$, their homomorphic product C_{mul} is computed as $C_{-1}^{h-1} \cdot C_1 \cdot C_2 \cdots C_h$, where the C_{-1}^{h-1} factor is due to the fact that each homomorphic multiplication requires a multiplication by C_{-1}. Then, the decryption procedure of C_{mul} computes the octonion $a_{mul} = \phi^{-1}(\mathbb{1} \cdot M \cdot C_{mul} \cdot M^{-1})$, deriving the plaintext value as $a_{mul} \cdot v^T$, where $v \in V^{\perp}$ is the vector appended to the secret key. Due to the homomorphic property, the octonion a_{mul} can also be written as the octonion product among the octonions $a_j = \phi^{-1}(\mathbb{1} \cdot M \cdot C_j \cdot M^{-1})$, $j \in \{-1, 1, 2, \dots, h\}$; i.e., $a_{mul} = a_{-1}^{h-1} * (a_1 * (a_2 * (\dots * a_h) \dots))$, where $a_{-1} = (-i + r_{-1} z)$ and $a_j = m_j i + r_j z, j \in \{1, 2, \dots, h\}$, with $m_j \in \mathbb{Z}_N$ being the plaintext of C_j, $r_{-1}, r_1, \dots, r_j \in \mathbb{Z}_N$ being the random values employed to construct C_j and z being the generator of the subspace V.

Applying Statement A.1 to a_{mul}, we obtain:

$$\begin{aligned} a_{mul} &= \left(i^{2h-1}(-1)^{h-1} \prod_{j=1}^{h} m_j \right) + R_z z + R_{iz}(i * z) \\ &+ R_{zi}(z * i) + R_{izi}(i * z * i) + R_{ziz}(z * i * z) \\ &= \left(i \prod_{j=1}^{h} m_j \right) + R_z z + R_{iz}(i * z) + R_{zi}(z * i) \\ &+ R_{izi}(i * z * i) + R_{ziz}(z * i * z) \end{aligned} \tag{A.8}$$

which follows from $i^{2h-1}(-1)^{h-1} = i^{2h-1}(i^2)^{h-1} = i^{2h-1} i^{2h-2} = i^{4h-3} = i^{4(h-1)+1} = i(i^4)^{h-1} = i(1)^{h-1} = i$. Eq. (A.8) shows that the additional octonion terms besides the product of the plaintext values $\left(i \prod_{j=1}^{h} m_j \right)$, which need to be removed to preserve decryption correctness, are equivalent to the sum of five octonions of the form reported in Eq. (A.4) regardless of the number of homomorphic multiplications performed among the ciphertexts. Therefore, in the last step of decryption, we can perform an inner product between a_{mul} and a vector $v \in V^{\perp}$ that is orthogonal to these five octonions. To find this vector $v = [v_1, 1, v_3, v_4, v_5, v_6, v_7, v_8]$, given N and $z = [z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8]$, the generator of the isotropic subspace V, we need to solve the following system of five equations in seven unknowns (the components of the vector v) over $\mathbb{Z}_N$:

$$\begin{cases} z \cdot v^T & \equiv_N 0 \\ (i * z) \cdot v^T & \equiv_N 0 \\ (z * i) \cdot v^T & \equiv_N 0 \\ (i * z * i) \cdot v^T & \equiv_N 0 \\ (z * i * z) \cdot v^T & \equiv_N 0 \end{cases} \tag{A.9}$$

where $\equiv_N$ denotes the equivalence relationship over $\mathbb{Z}_N$. By adding member-wise the first and third equalities in Eq. (A.9), we obtain the following constraint on the elements of v and z:

$$(2v_1 \times (-z_2) + 2v_2 \times z_1) \equiv_N 0 \;\Rightarrow\; (v_1 \times (-z_2) + z_1) \equiv_N 0 \;\Rightarrow z_1 \equiv_N v_1 \times z_2$$

By subtracting member-wise the first and fourth equalities in Eq. (A.9), we obtain a further constraint on the elements of v and z:

$$(2v_1 \times z_1 + 2v_2 \times z_2) \equiv_N 0 \;\Rightarrow\; (v_1 \times z_1 + z_2) \equiv_N 0$$

By replacing the first constraint in the second one, it is possible to find the following necessary condition on v_1 to solve the simultaneous set of equalities in Eq. (A.9):

$$(v_1 \times z_1 + z_2) \equiv_N 0 \;\Rightarrow\; z_2 \times (v_1^2 + 1) \equiv_N 0.$$

There are three possible cases to be analyzed to solve this equation:

1. $z_2 = 0$. In this case, any integer v_1 would satisfy this equation. Moreover, the first of the aforementioned constraints allows to infer that also $z_1 = 0$, since $z_1 \equiv_n v_1 \times z_2 \equiv_N 0$.
2. z_2 is coprime with N, and thus it is invertible. Therefore, we have:

   $$z_2 \times (v_1^2 + 1) \equiv_N 0 \Rightarrow v_1^2 + 1 \equiv_N 0 \Rightarrow v_1^2 \equiv_N -1$$

   This equation has a solution if and only if there is a square root of -1 in $\mathbb{Z}_N$, that is an integer ι_N such that $\iota_N^2 = -1 \bmod N$.
3. z_2 is not coprime with N. We denote a generic common divisor between N and z_2 as g. Now, since $z_2 \times (v_1^2 + 1) \equiv_N 0$, then $z_2 \times (v_1^2 + 1)$ must be a multiple of N; since z_2 is a multiple of g, then $v_1^2 + 1$ must be a multiple of $\frac{N}{g}$. Therefore, $v_1^2 + 1 \equiv_{\frac{N}{g}} 0$, which again has a solution if and only if there is a square root of -1 in $\mathbb{Z}_{\frac{N}{g}}$. Obviously, if there are square roots of -1 in $\mathbb{Z}_N$, $v_1 = \iota_N$ is a solution also in this case.

The sets of constraints ensuring that it is always possible to find a vector v allowing a correct decryption are thus summarized as follows.

Statement A.2. *The system in Eq.* (A.9) *may exhibit a solution $v = [v_1, 1, v_3, v_4, v_5, v_6, v_7, v_8]$ only if at least one of these three set of constraints on the modulus N of the ring $\mathbb{Z}_N$ and on the generator $z = [z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8]$ of the monodimensional isotropic space V are satisfied:*

1. *$z_1 = 0$ and $z_2 = 0$, $N \in \mathbb{N} \setminus \{0, 1\}$.*
2. *N is a composite integer for which square roots of -1 exist in $\mathbb{Z}_N$, denoted by ι_N. z_1 must be chosen as $z_1 \equiv_N \iota_N z_2$, while $z_2 \in \mathbb{Z}_N$.*
3. *N is a composite integer such that, for at least one of its divisors g, square roots of -1 exist in $\mathbb{Z}_{\frac{N}{g}}$, and are denoted by $\iota_{\frac{N}{g}}$. z_2 must be a multiple of g and $z_1 \equiv_N \iota_{\frac{N}{g}} z_2$.*

Willing to avoid the introduction of strong constraints for the choice of z, as it contains the randomness employed to hide the plaintext, we pick the second set of constraints in our instantiation of `OctoM`. We therefore need to pick a composite N as a modulus such that square roots of -1 exist in $\mathbb{Z}_N$. The specific constraints on the composite integer N that allow to meet this requirement are detailed in the following theorem:

Theorem A.2. *Given a composite integer $N = \prod_{l=1}^{h} p_l$ with h prime factors $p_l, l \in \{1, \ldots, h\}$, square roots of -1 exist in $\mathbb{Z}_N$ if and only if $p_l \bmod 4 = 1$ for all prime factors $p_l, l \in \{1, \ldots, h\}$.*

Proof. Recall that, by Euler's criterion, for a prime p and an integer a coprime with p, if the square root of a exists in $\mathbb{Z}_p$ then $a^{\frac{p-1}{2}} = 1$, otherwise $a^{\frac{p-1}{2}} = -1$. Consider the CRT decomposition of $\mathbb{Z}_N$: for $N = \prod_{l=1}^{h} p_l$, $\mathbb{Z}_N$ and $\mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \cdots \times \mathbb{Z}_{p_h}$ are isomorphic. Therefore, an equation has a solution in $\mathbb{Z}_N$ if and only if it has a solution on each of the rings $\mathbb{Z}_{p_l}$. Hence, the equation $x^2 \equiv_N -1$, whose solution is a square root of -1 in $\mathbb{Z}_N$, has a solution if and only if each equation $x^2 \equiv_{p_l} -1$ has a solution. In particular, each of these equations has a solution if and only there is a square root of -1 in $\mathbb{Z}_{p_l}$, which can be determined with Euler's criterion. For each prime number p_l, we compute $a^{\frac{p-1}{2}}$, with $a = -1$:

$$(-1)^{\frac{p_l-1}{2}} = \begin{cases} 1 & \text{if } \frac{p_l-1}{2} \text{ is even} \\ -1 & \text{if } \frac{p_l-1}{2} \text{ is odd} \end{cases}$$

Therefore, there is a square root of -1 if and only if $\frac{p_l-1}{2}$ is even, which implies that $p_l \bmod 4 = 1$. $\square$

Summing up, to make OctoM a multiplicatively homomorphic scheme, we extend the key generation algorithm computing the key $k = (M, \phi, V, v)$ and the evaluation key $evk = (N, C_{-1})$ with the following additional constraints:

1. Each prime factor of the composite integer N used to perform the modular operations must be chosen such that it is congruent to one modulo four (i.e., $N = \prod_{l=1}^{h} p_l$, with $p_l \bmod 4 = 1, l \in \{1, \ldots, h\}$).
2. The generator $z = [z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8]$ of the totally isotropic subspace V must be chosen such that $z_1 = \iota_N \cdot z_2$, where ι_N is a square root of -1 in $\mathbb{Z}_N$.
3. The octonion $v \in V^{\perp}$ must be obtained by solving the simultaneous set of equalities in Eq. (A.9).

We experimentally verify in our pilot implementation the evaluation correctness of the homomorphic multiplication with such a modified KEYGEN procedure, hereby making OctoM a FHE scheme.

2 Security Proof for Our PPSS Protocol

In this section, we prove Thm. 3.1 by showing the existence of a simulator $\mathcal{S}$ which interacts with any semi-honest adversary $\mathcal{A}$, according to the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment of Def. 3.6, to produce a transcript for this experiment which is computationally indistinguishable from the transcript of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, where $\mathcal{A}$ interacts with a client through our actual PPSS protocol. For the sake of clarity, in the following we denote all the variables involved in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment with a superscript *Id* (e.g., $\langle \mathbf{D} \rangle^{Id}$ denotes the privacy-preserving representation of the document collection $\langle \mathbf{D} \rangle$ computed by the simulator $\mathcal{S}_{\mathbf{D}}$). We also assume that the alphabet Σ for the documents in $\mathbf{D}$ is publicly known.

Simulator Construction. We now show how to construct the simulator $\mathcal{S}$. Specifically, for a document collection $\mathbf{D}$ of z documents $D_1, \ldots, D_z$ and a string q, $\mathcal{S}$ is realized by constructing two simulators $\mathcal{S}_{\mathbf{D}}$ and $\mathcal{S}_q$. The former employs the leakage $\mathcal{L}_{\mathbf{D}}$ to build a privacy-preserving representation $\langle \mathbf{D} \rangle^{Id}$ that must be computationally indistinguishable from the privacy-preserving representation $\langle \mathbf{D} \rangle$ computed by the client in our PPSS protocol. The latter simulator employs both the leakage $\mathcal{L}_{\mathbf{D}}$ and $\mathcal{L}_q$ to build a trapdoor $\langle q \rangle_j^{Id}$, $j{\leq}1{\leq}w$, for each of the w rounds of the QUERY procedure looking up the string q; all these trapdoors must be computationally indistinguishable from the trapdoors constructed by the client in the w rounds of our PPSS protocol.

- $\mathcal{S}_{\mathbf{D}}$. Given the leakage $\mathcal{L}_{\mathbf{D}} = \sum_{i=1}^{z}(\textsc{len}(D_i) + 1) = n$, the simulator constructs two arrays A^{Id} and Suf^{Id}. The former (resp. the latter) array has $\lceil \frac{n+1}{P} \rceil$ (resp. $n+1$) entries, each containing a randomly generated bit string with the same size of an entry of the ABWT full-text index constructed with sample period P (resp. the suffix array), that is $\Theta(|\Sigma|\log(n) + P\log(|\Sigma|)$ (resp. $\Theta(\log(n))$) bits. The simulator outputs the privacy-preserving representation $\langle \mathbf{D} \rangle^{Id} = (A^{Id}, \mathit{Suf}^{Id})$
- $\mathcal{S}_q$. Given the leakages $\mathcal{L}_{\mathbf{D}}$, $\mathcal{L}_q = (\textsc{len}(q), b, |O_{\mathbf{D},q}|)$ and the public modulus N for the LFAHE DJ scheme employed by the client in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, the simulator computes the values $t_A = \lceil \log_b(\lceil \frac{n+1}{P} \rceil) \rceil$ and $t_{\mathit{Suf}} = \lceil \log_b(\lceil \frac{n+1}{o_q} \rceil) \rceil$, where $o_q = |O_{\mathbf{D},q}|$. Then, the simulator constructs $m = \textsc{len}(q)$ trapdoors $\langle q \rangle_1^{Id}, \ldots, \langle q \rangle_m^{Id}$ as follows. Each trapdoor is an array with $b \cdot t_A$ elements, where the first b entries are integers randomly sampled in $\mathbb{Z}^*_{N^2}$, the subsequent b entries are integers randomly sampled in $\mathbb{Z}^*_{N^3}$, and so on; in general, the j-th entry contains an integer randomly sampled in $\mathbb{Z}^*_{N^{\lceil \frac{j}{b} \rceil + 1}}$. Then, the simulator generates two trapdoors $\langle q \rangle_{m+1}^{Id}, \langle q \rangle_{m+2}^{Id}$, where each trapdoor is an array with $b \cdot t_{\mathit{Suf}}$ elements constructed in the same manner as the previous m trapdoors (i.e., the j-th entry contains an integer randomly sampled in $\mathbb{Z}^*_{N^{\lceil \frac{j}{b} \rceil + 1}}$).

We now prove that, for any probabilistic polynomial time adversary $\mathcal{A}$, the transcript of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment is computationally indistinguishable from the transcript of the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment when the simulator $\mathcal{S}$ we have just constructed is employed. Specifically, we analyze each step of the two experiments and we show that the adversary cannot distinguish the simulator from the client of our PPSS protocol. In both the experiments, the adversary initially chooses a document collection $\mathbf{D}$ of z documents over a publicly known alphabet Σ. In the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, $\mathbf{D}$ is sent to the client, which constructs a privacy-preserving representation $\langle \mathbf{D} \rangle$ by running the SETUP procedure of our PPSS protocol; specifically, $\langle \mathbf{D} \rangle$ is composed by two cell-wise encrypted arrays $\langle \mathbf{D} \rangle = (\langle A_P \rangle, \langle \mathit{Suf} \rangle)$ with, respectively, $\lceil \frac{n+1}{P} \rceil$ and $n+1$ elements. Conversely, in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment, the simulator $\mathcal{S}_{\mathbf{D}}$ obtains the leakage $\mathcal{L}_{\mathbf{D}}$ and constructs the privacy-preserving representation $\langle \mathbf{D} \rangle^{Id}$ as two arrays $A^{Id}, \mathit{Suf}^{Id}$ whose size is the same as, respectively, $\langle A_P \rangle$ and $\langle \mathit{Suf} \rangle$ ones. The semantic security of the scheme $\mathcal{E}$ employed to encrypt $\langle A_P \rangle$ and $\langle \mathit{Suf} \rangle$ in our PPSS protocol guarantees that a ciphertext computed by $\mathcal{E}$.`Enc` is computationally indistinguishable from a random bit string with the same number of bits of the ciphertext, which implies that each entry of $\langle A_P \rangle$ (resp. $\langle \mathit{Suf} \rangle$) is computationally indistinguishable from each entry of A^{Id} (resp. Suf^{Id}); therefore, the two privacy-preserving representations $\langle \mathbf{D} \rangle$ and $\langle \mathbf{D} \rangle^{Id}$ are computationally indistinguishable too.

After receiving the privacy-preserving representations $\langle \mathbf{D} \rangle$ and $\langle \mathbf{D} \rangle^{Id}$, the adversary chooses a string q_1. In the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, the string q_1 is sent to the client, which employs the QUERY procedure of our PPSS protocol to find all the positions of the occurrences of q_1 in $\mathbf{D}$. In each of the $m{+}2$ rounds of the QUERY procedure, the client employs the TRAPDOOR procedure to generate a trapdoor $\langle q_1 \rangle_j,\ j{\leq}1{\leq}m{+}2$, which corresponds to a trapdoor in the Lipmaa's PIR protocol. In the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment, the simulator $\mathcal{S}_{q_1}$ receives the leakage $\mathcal{L}_{q_1}$, which is employed to build a trapdoor $\langle q_1 \rangle_j^{Id},\ j{\leq}1{\leq}m{+}2$ for each of the $m{+}2$ rounds. The semantic security of the LFAHE DJ scheme guarantees that a ciphertext computed by the encryption procedure with length l is computationally indistinguishable from a random integer in $\mathbb{Z}^*_{N^{l+1}}$, which means that the set of trapdoors $\langle q_1 \rangle_j$ are computationally indistinguishable from the set of trapdoors $\langle q_1 \rangle_j^{Id}$, for all $j \in \{1, \ldots, m{+}2\}$.

Subsequently, in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ (resp. $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$) experiment, the trapdoor $\langle q_1 \rangle_j$ (resp. $\langle q_1 \rangle_j^{Id}$) generated by the client (resp. $\mathcal{S}_{q_1}$) in each of the $m{+}2$ rounds is received by the adversary, which employs the SEARCH procedure of Lipmaa's PIR protocol to compute a ciphertext $\langle res_j \rangle$ (resp. $\langle res_j \rangle^{Id}$). The semantic security of the LFAHE DJ scheme guarantees that all the intermediate values computed by each homomorphic operation of the SEARCH procedure in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment are computationally indistinguishable from the corresponding intermediate values in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment. Indeed, in the former experiment, given two ciphertext c_1 and c_2 in $\mathbb{Z}^*_{N^l}$ for the LFAHE DJ scheme, each *homomorphic addition* computes $c_{add} = c_1 \cdot c_2 \bmod N^l$, with c_{add} being a ciphertext in $\mathbb{Z}^*_{N^l}$; in the latter experiment, the homomorphic addition multiplies two random integers in $\mathbb{Z}^*_{N^l}$, obtaining a new random integer in $\mathbb{Z}^*_{N^l}$ which is computationally indistinguishable from c_{add}. Similarly, in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ experiment, given a ciphertext $c_1 \in \mathbb{Z}^*_{N^l}$ and a ciphertext $c_2 \in \mathbb{Z}^*_{N^{l+1}}$, each *hybrid homomorphic multiplication* computes $c_{hmul} = c_2^{c_1} \bmod N^{l+1}$, with c_{hmul} being a ciphertext in $\mathbb{Z}^*_{N^{l+1}}$; in the $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiment, each hybrid homomorphic multiplication computes the exponentiation between a random integer in $\mathbb{Z}^*_{N^{l+1}}$ and a random integer in $\mathbb{Z}^*_{N^l}$, obtaining a new random integer in $\mathbb{Z}^*_{N^{l+1}}$ which is computationally indistinguishable from c_{hmul}. Therefore, as the SEARCH procedure of Lipmaa's PIR performs only homomorphic operations, we conclude that all values (including the outcomes $\langle res_j \rangle$ and $\langle res_j \rangle^{Id}$) observed by the adversary throughout this computation in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}$ experiments are computationally indistinguishable. In conclusion, the adversary cannot distinguish an interaction with a legitimate client in our PPSS protocol from an interaction with the simulator $\mathcal{S}_{q_1}$ for the first query q_1.

We note that the same reasoning allows to prove that all the trapdoors and the intermediate values observed by the adversary in the subsequent $d - 1$ queries in the two experiments are computationally indistinguishable.

3 Security Proof for ObSQRE

In this section, we first prove that our DORAMs meet the security guarantees stated in Thm. 4.1 and then we show that ObSQRE achieves the security guarantees stated in Thm. 4.2. We remark that, although our security definitions are applicable to all the DORAMs and all the oblivious substring search algorithms discussed in this work, for the sake of conciseness, we actually prove the security guarantees only for the DORAM and for the substring search algorithm employed in the solution exhibiting the best performance in our experimental evaluation, that are Circuit DORAM and ABWT based substring search algorithm, respectively.

3.1 DORAM Security Proof

To prove that Circuit DORAM exhibits the security guarantees stated in Thm. 4.1, we show the construction of a simulator $\mathcal{S}$ that makes the transcript of the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment computationally indistinguishable from the $\mathbf{Real}_{\rho,\mathcal{A}}$ one.

Simulator $\mathcal{S}_{Init}$. This algorithm, given $\mathcal{L}_{Init}$ as input, randomly samples M blocks $D_i^{\mathcal{S}}$ of B bits, $i \in \{1, \ldots, M\}$. The traces $\mathcal{T}_{Init}$ observed by the adversary in the two experiments are indistinguishable: indeed, the DORAMs are constructed with similar parameters in both the experiments, and the INIT procedure accesses the same blocks in both the experiments (as paths to be evicted are chosen according to a deterministic schedule). Since the blocks inserted in the DORAMs are encrypted with a semantically secure cipher, it is not possible to distinguish the blocks with random data employed in the $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiment from the blocks with actual data of the $\mathbf{Real}_{\rho,\mathcal{A}}$ one. The result res_0 of the INIT procedure is the same in both experiments, as the tampering of the tree is detected independently from the data stored inside the DORAM. Besides such tampering, there are no other adversarial behaviors that may alter the result of the computation, since any such tampering involves code and data stored inside the SGX enclave, whose integrity is ensured by SGX security guarantees.

Simulator $\mathcal{S}_{\mathbf{Acc,i}}, \mathbf{i} \in \{\mathbf{1}, \ldots, \mathbf{d}\}$. This simulator simply chooses at random the block id $bid_i^{\mathcal{S}}$ to be accessed by the DORAM. We now show that the traces $\mathcal{T}_{Acc,i}$ of our Circuit DORAM observed by the adversary in both the experiments are computationally indistinguishable. We start by proving the following claim about the code and data access patterns of the client algorithms in our Circuit DORAM:

Theorem A.3. *The code and data access patterns ($Code_{AP}$ and $Data_{AP}$) of our Circuit DORAM client in the* ACCESS *procedure are independent from the block id bid given as input to the procedure*

Proof. The ACCESS procedure of the DORAM has two main phases: the former retrieves from the position map the leaf id lid corresponding to block bid, replacing lid with a new random leaf id lid' in the corresponding entry of the position map; the latter employs the FINDBLOCK and EVICTION procedures to retrieve the block with id bid from the DORAM. We first prove our claim for these two procedures; then, we prove it also for the first phase when the position map is recursively stored in several DORAMs.

FINDBLOCK. The DORAM client executes the FINDBLOCK procedure reported in Alg. 4.1. Both the control flow and the memory locations accessed by this procedure are clearly independent from the block id bid: indeed, the former depends only on parameters of the DORAM known to the adversary, while all conditionally dependent memory accesses are performed through oblivious operations.

EVICTION. This procedure, reported in Alg. 4.4, is executed over two paths, chosen by the client with a deterministic schedule known to the adversary and independent from bid; furthermore, as the control flow of this procedure depends only on parameters of the DORAM known to the adversary and all the conditionally dependent memory accesses are performed through oblivious operations, both code and data access patterns of this procedure are independent from the block id bid.

Recursive Position Map. We recall that the position map of the DORAM is stored in $\mathrm{O}(\log_C(n))$ DORAMs of increasing size, and the client stores only the position map of the smallest among these DORAMs. The algorithm to retrieve the leaf id corresponding to block bid has $\mathrm{O}(log_C(n))$ iterations; in each of them, the algorithm accesses one of the DORAMs storing the position map, hinging upon FINDBLOCK and EVICTION procedures: as we have just shown, their operations are independent from bid. The algorithm, after retrieving a block from these DORAMs, performs a linear sweep over such block; this block is a small array with $\mathrm{O}(C)$ entries, and each of them is obliviously swapped with a target memory location through the OBLSWAP operation. Therefore, the sweep over the block does not depend on bid. No other operations are performed in each iteration of the algorithm, thus making the control flow and the memory access pattern of the retrieval of the leaf id lid corresponding to the block with id bid independent from bid. □

The claim in Thm. A.3 implies that there is no difference on the access patterns observed by the adversary throughout the ACCESS procedure in the $\mathbf{Real}_{\rho,\mathcal{A}}$ and $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiments. Regarding $Data_{srv}$, that is the information sent outside the enclave from the DORAM client, we observe that this is limited to the leaf ids of the paths being fetched or evicted, and the blocks of these paths written back to the DORAM tree. The paths to be fetched in our DORAMs are chosen in the same way as in the corresponding ORAM; thus, the leaf ids of these paths are distributed as in the corresponding ORAM. Since leaf ids of fetched paths in Circuit ORAM are uniformly distributed, independently from the accessed blocks, then the distribution of the leaf ids fetched by our DORAM is uniform in both the $\mathbf{Real}_{\rho,\mathcal{A}}$ and $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiments. Regarding the paths being evicted, in all our DORAMs they are chosen according to a deterministic schedule that depends only on the eviction period A, which is the same in both experiments. Finally, since the blocks are encrypted with a semantically secure scheme, the paths being written back after fetch or eviction appear as indistinguishable random data in both the experiments. In conclusion, the traces $\mathcal{T}_{Acc,i}$, $i \in \{1, \ldots, d\}$, are computationally indistinguishable between the $\mathbf{Real}_{\rho,\mathcal{A}}$ and $\mathbf{Ideal}_{\rho,\mathcal{A},\mathcal{S}}$ experiments.

Regarding the result res_i of the ACCESS procedure, $i \in \{1, \ldots, d\}$, the integrity check mechanism ensures that any tampering on the path fetched from the DORAM tree is detected in both experiments. Conversely, in case the adversary decides to tamper with a randomly chosen path before knowing which path will be fetched, the results between the two experiments may differ; nonetheless, as the adversary cannot guess with other than uniform probability the path being fetched in the ACCESS procedure in both experiments, the statistical distribution of tampering detection is equivalent to the distribution of correctly guessing the path being fetched, which is uniform in both experiments. The adversary has no other ways to tamper with data and computation, as all the other operations are performed inside the enclave over the data stored inside the enclave, thus proving that the results res_i, $i \in \{1, \ldots, d\}$, are computationally indistinguishable between the experiments.

3.2 ObSQRE Security Proof

We now prove that ObSQRE achieves the security guarantees reported in Thm. 4.2, assuming that a DORAM fulfilling the privacy guarantees outlined in Thm. 4.1 is employed in our ABWT based oblivious substring search algorithm. To this extent, we describe the simulator $\mathcal{S}$ and we show that the output of the $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$ experiment is computationally indistinguishable from the output of the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ one.

Simulator $\mathcal{S}_0$. This simulator, upon receiving $\mathcal{L}_{Setup}$ and $\mathcal{L}_{Load}$, constructs a document collection $\mathbf{D}^{\mathcal{S}}$ over the publicly known alphabet Σ by randomly sampling z strings whose lengths sum up to n. Then, the simulator computes the ABWT-based full-text index and the SA from $\mathbf{D}^{\mathcal{S}}$ and encrypts them with an AEAD scheme, obtaining the encrypted index $\mathcal{I}^{\mathcal{S}}$. This index has the same size of the index $\mathcal{I}$ computed from the document collection $\mathbf{D}$ chosen by the adversary; furthermore, each of its entries are encrypted with a semantically secure scheme, in turn making the indexes $\mathcal{I}, \mathcal{I}^{\mathcal{S}}$ (and thus the traces $\mathcal{T}_{Setup}$ in the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ and $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$ experiments) computationally indistinguishable. The traces $\mathcal{T}_{Load}$ are also computationally indistinguishable in both the experiments. Indeed, in the LOAD procedure, $\mathcal{I}^{\mathcal{S}}$ is decrypted, and then the ABWT and the SA are inserted into the DORAM through the INIT operation. The security guarantees of the DORAM ensures that this operation leaks only the number and the size of the DORAM blocks: their number is proportional, for both the ABWT and the SA, to n, which is the same in both experiments; the size of each block corresponds to the size of each entry of the ABWT and the SA, respectively, which are already known to the adversary. Finally, the security guarantees of the DORAM ensures that INIT procedure is secure against any tampering to the DORAM tree, while the AEAD scheme guarantees that any tampering on the encrypted indexes $\mathcal{I}, \mathcal{I}^{\mathcal{S}}$ is detected in the LOAD procedure, hence making the results res_0 equivalent in both experiments.

Simulator $\mathcal{S}_{\mathrm{i}}, \mathbf{i} \in \{\mathbf{1}, \ldots, \mathbf{d}\}$. This simulator, given $\mathcal{L}_{Query,i}$, chooses a random string $q_i^{\mathcal{S}}$ of length m_i and sets $occ_i^{\mathcal{S}} = occ_i$. The QUERY procedure then, upon receiving $q_i^{\mathcal{S}}$ and $occ_i^{\mathcal{S}}$, employs the oblivious backward search algorithm with ABWT based oblivious RANK procedure. The number of iterations of backwards search depends only on m_i and occ_i in both the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ and the $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$ experiments. The linear sweeps over the dictionary `Count` adds to the trace $\mathcal{T}_{Query}$ only its size $|\Sigma|$, as each entry is involved in an oblivious write. The oblivious RANK procedure, outlined in Alg. 4.5, retrieves a block from the DORAM, whose security guarantees ensures that no information is leaked during the ACCESS operation. After retrieving such block, the RANK procedure obliviously sweeps over this block, an operation that reveals only the block size, which is already known from $\mathcal{T}_{Load}$. Concerning the result of the query, all the operations, except for the DORAM ACCESS, are performed inside the SGX enclave, where any code and data tampering is prevented. As the security guarantees of DORAM ensures that accesses are secure against any tampering strategy, then the results of the queries are computationally indistinguishable in both the $\mathbf{Real}_{\mathcal{P},\mathcal{A}}$ and the $\mathbf{Ideal}_{\mathcal{P},\mathcal{A},\mathcal{S}}$ experiments.

4 Oblivious EARLYRESHUFFLE Analysis

In this section, we prove that the strategy employed by the EARLYRESHUFFLE procedure of our Ring DORAM places Z blocks out of the $Z{+}D$ slots available in a bucket uniformly at random. To this extent, we define the event $\mathcal{E}_{i,j}$, $i{\in}\{1, \ldots, Z\}$, $j{\in}\{0, \ldots, Z{+}D{-}1\}$, which is verified if the slot j of the bucket is full after the EARLYRESHUFFLE procedure has placed the i-th block in the bucket. Similarly, we define the event $\mathcal{B}_{i,j}$, $i{\in}\{1,\ldots,Z\}$, $j{\in}\{0,\ldots,Z{+}D{-}1\}$, which is verified if the i-th block placed by EARLYRESHUFFLE is assigned to the slot j of the bucket. Clearly, the i-th block is assigned to the slot j if and only if this slot is chosen in the i-th iteration and it is never chosen in all previous iterations, i.e, $\mathcal{B}_{i,j} = \bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j} \wedge \mathcal{E}_{i,j}$. The probability of the event $\mathcal{B}_{ij}$ can be thus computed as:

$$\begin{aligned}
\mathtt{Pr}(\mathcal{B}_{i,j}) &= \mathtt{Pr}(\bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j} \wedge \mathcal{E}_{ij}) = \mathtt{Pr}(\mathcal{E}_{i,j}|\bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{hj})\mathtt{Pr}(\bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j}) \\
&= \mathtt{Pr}(\mathcal{E}_{i,j}|\bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j})\mathtt{Pr}(\neg\mathcal{E}_{i-1,j}|\mathtt{Pr}(\bigwedge_{h=1}^{i-2} \neg\mathcal{E}_{h,j}))\mathtt{Pr}(\bigwedge_{h=1}^{i-2} \neg\mathcal{E}_{h,j}) \qquad \text{(A.10)} \\
&= \mathtt{Pr}(\mathcal{E}_{i,j}|\bigwedge_{h=1}^{i-1} \neg\mathcal{E}_{h,j})\mathtt{Pr}(\neg\mathcal{E}_{i-1,j}|\mathtt{Pr}(\bigwedge_{h=1}^{i-2} \neg\mathcal{E}_{h,j})\cdots\mathtt{Pr}(\neg\mathcal{E}_{1,j})
\end{aligned}$$

We now compute each of these probabilities. $\mathtt{Pr}(\neg\mathcal{E}_{1,j})$ is the probability that the slot j is not chosen in the first iteration; since each of the $Z{+}D$ slots may be chosen with uniform probability, $\mathtt{Pr}(\neg\mathcal{E}_{1,j}) = \frac{Z+D-1}{Z+D}$. $\mathtt{Pr}(\neg\mathcal{E}_{h,j}|\bigwedge_{k=1}^{h-1} \neg\mathcal{E}_{k,j})$ is the probability that the slot j is not chosen among the $Z + D - h + 1$ ones still available in the h-th iteration; since each of them may be chosen with uniform probability, then $\mathtt{Pr}(\neg\mathcal{E}_{h,j}|\bigwedge_{k=1}^{h-1} \neg\mathcal{E}_{k,j}) = \frac{Z+D-h}{Z+D-h+1}$. Finally, $\mathtt{Pr}(\mathcal{E}_{i,j}|\bigwedge_{z=1}^{i-1} \neg\mathcal{E}_{z,j})$ is the probability that the slot j is chosen in the i-th iteration; as the slot is chosen uniformly at random among $Z + D - i + 1$ ones, then $\mathtt{Pr}(\mathcal{E}_{i,j}|\bigwedge_{z=1}^{i-1} \neg\mathcal{E}_{z,j}) = \frac{1}{Z+D-i+1}$. Substituting these probabilities in Eq. (A.10), we obtain:

$$\mathtt{Pr}(\mathcal{B}_{i,j}) = \frac{1}{Z+D-i+1}\prod_{h=1}^{i-1}\frac{Z+D-h}{Z+D-h+1} = \frac{1}{Z+D}$$

Since the analysis may be repeated for each slot j and for each of the Z blocks, we conclude that each block is placed with uniform probability over all the $Z{+}D$ slots of the bucket.