Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

POLITECNICO
MILANO 1863

ObSQRE: efficient full-text index for oblivious substring search queries with Intel SGX

Relatore:	Prof. Gerardo PELOSI
Correlatore:	Dott. Nicholas MAINARDI

Tesi di Laurea di:
Davide SAMPIETRO Matr. 877938

Anno Accademico 2018–2019

To my father and my mother

Ringraziamenti

In primo luogo vorrei esprimere la mia gratitudine per le persone senza le quali questo lavoro non sarebbe stato realizzabile. Gerardo Pelosi ed Alessandro Barenghi hanno messo a disposizione il loro tempo e la loro esperienza per indirizzare questo progetto nella direzione più fruttuosa, mentre Nicholas è sempre stato disponibile nel fornirmi l'aiuto e gli spunti riflessivi che hanno permesso di superare le difficoltà più spinose di questo percorso. Grazie a loro ho affrontato problematiche da una prospettiva totalmente diversa da quella usuale, cosa che mi ha arricchito e incentivato a dare il massimo.

Un ringraziamento speciale va alla mia famiglia, a mio padre e mia madre, che mi hanno sostenuto ed accompagnato durante l'intero percorso universitario, passo dopo passo. Se sono arrivato fino a questo punto lo devo solo a loro: innanzitutto per la loro dedizione al dovere, tenacia e forza di volontà, che mi hanno trasmesso fin da sempre e che mi ha consentito di andare avanti anche nei momenti più difficili. Oltre a spronarmi nei periodi di demotivazione, hanno sempre tentato di smorzare la mia indole pessimista, facendo il possibile per alleggerirmi dalle troppe responsabilità che tendo ad assumermi.

Una menzione di eccezione va ai miei amici di sempre, Gabriele, Francesca, Alessio, Pietro, Fabio e Marco, che non mi hanno disconosciuto nonostante la mia costante assenza durante le sessioni di esame. Loro sono le uniche persone che riescono a strapparmi un sorriso anche nelle situazioni più stressanti. Vorrei anche ringraziare Gianluca, che è stata la presenza più incisiva durante questi anni trascorsi a Milano.

Abstract

The problem of finding the positions of the repetitions of a string within a text, known as substring search, recurs in many application scenarios. When the size of the text is huge, the construction of a full-text index is the only viable solution since it allows to perform substring search queries in sublinear time. In this scenario, it may be unconvenient to store the index locally, in turn requiring to outsource both the data and the computation to the cloud. This may become an issue when dealing with confidential information, e.g. biological data of a patient who resorts to personalized medicine, since an untrusted cloud service provider may gain illicit benefits from such confidential data. Secure enclaves allow to instantiate trusted and authenticated execution environments on untrusted machines, preventing any rogue access to their internal content. While they allow to securely handle confidential data in plaintext, the most widespread implementation of enclaves, Intel SGX, is prone to side channel attacks that are especially effective in leaking sensitive information. In this work, we propose Oblivious Substring Query on Remote Enclave (ObSQRE), a novel substring search protocol that implements oblivious full-text indices to solve the private substring search problem in the outsourcing scenario. It prevents an adversary with root privileges on an untrusted machine from inferring the structure of the text as well as the correlation between queries. It encompasses three different substring search algorithms that rely on the ORAM cryptographic primitive to prevent any information leakage. To the best of our knowledge, Oblivious Substring Query on Remote Enclave (ObSQRE) is the first to directly address the remote private substring search problem exploiting secure enclaves. ObSQRE exhibits practical performance, being able to find occurrences of a protein ($\sim$ 3000 nucleotides) within 32 MB of genomic data in only 500 ms.

Sommario

La ricerca di sottostringhe all'interno di un testo più ampio è un problema comune in diversi scenari. Quando la dimensione del testo è considerevole, opportune strutture di indicizzazione consentono di cercare sottostringhe con complessità sublineare nella lunghezza del testo; date le dimensioni usuali di un indice, potrebbe essere dispendioso memorizzarlo ed interrogarlo su una macchina desktop. Una possibile soluzione consiste nell'esternalizzare sia l'indice che la ricerca delle sottostringhe presso un fornitore di servizi di cloud computing. Quando i dati contenuti nel testo sono sensibili, come per esempio quelli del genoma di un paziente che si sottopone a trattamenti di medicina personalizzata, affidare la gestione delle informazioni ad un soggetto terzo presenta criticità legate alla privacy dei dati. In questo contesto, le enclavi consentono di eseguire le porzioni sensibili di un programma in un'area di memoria inaccessibile dall'esterno, consentendo di operare su dati in chiaro senza compromettere la loro segretezza. Tuttavia l'implementazione di enclavi più diffusa, Intel SGX, è vulnerabile ad attacchi a side channel, che sono in grado di estrarre il contenuto di enclavi che non adottano contromisure specifiche. In questo lavoro presentiamo Oblivious Substring Query on Remote Enclave (ObSQRE), un nuovo protocollo per la ricerca di sottostringhe basato su indici testuali cosiddetti *oblivious*, che garantiscono sia la segretezza del testo che delle sottostringhe cercate, nascondendo anche le loro similarità. ObSQRE fornisce tre distinti algoritmi di ricerca, che sfruttano le ORAM per nascondere le informazioni recuperabili tramite side channels. Secondo la nostra indagine, ObSQRE è la prima soluzione per la ricerca di sottostringhe in maniera oblivious basata su enclavi. I risultati sperimentali mostrano l'efficienza di ObSQRE in scenari reali, dato che è possibile cercare una proteina ($\sim$ 3000 nucleotidi) in un genoma di 32 MB in circa 500 ms.

Estratto in Italiano

La nascita e la diffusione di servizi di *cloud computing*, che mettono a disposizione sia spazio di archiviazione remoto che infrastrutture per l'esecuzione di carichi di lavoro intensivi, ha offerto una valida alternativa all'allestimento di cluster privati per la gestione e l'elaborazione dell'informazione. Nel momento in cui la quantità di dati da processare diventa ingente, l'affitto di risorse e macchine remote diventa la soluzione più conveniente. Infatti, la messa a punto di server privati richiede un investimento considerevole sia per l'acquisto delle componenti hardware sia per i costi di gestione, che includono la necessità di personale specializzato per la manutenzione dell'infrastruttura. D'altro canto, i *cloud provider* offrono elevate garanzie di affidabilità e tolleranza ai guasti come parte integrante dei loro servizi.

Tuttavia, quando un client ha la necessità di elaborare una mole ingente di dati confidenziali, potrebbero sorgere seri problemi riguardanti la privacy. Infatti, il cloud provider è un'entità esterna che potrebbe trarre beneficio dall'impiego fraudolento delle informazioni che risiedono sui suoi server, pertanto è buona norma considerarlo una potenziale minaccia. Sebbene la cifratura dei dati ne protegga il contenuto, essa rappresenta un ostacolo per la loro elaborazione: gli algoritmi di cifratura standard, infatti, impediscono di utilizzare i dati per successive operazioni prima che essi vengano decriptati. In casi estremi in cui le performance non sono un problema, il client potrebbe accedere ai dati, memorizzati in remoto e processarli localmente. Tuttavia è stato dimostrato che il server può estrarre informazioni confidenziali ispezionando l'ordine in cui i dati sono richiesti dal client, definito *pattern di accesso*, se esso dispone di abbastanza informazioni di dominio [5, 19, 29]. Pur esistendo primitive quali la crittografia omomorfa, o Fully Homomorphic Encryption (FHE), che permettono di operare direttamente su dati cifrati, esse comportano un costo computazionale così alto che non possono essere impiegate nella pratica.

D'altro canto, in alcuni scenari applicativi, tecniche puramente crittografiche consentono di ottenere tempi di elaborazione ragionevoli garantendo alti margini di confidenzialità. In particolare, protocolli di Symmetric Searchable Encryption (SSE) consentono di effettuare ricerche (o *query*) su dati cifrati senza rivelare né il contenuto dei dati remoti, né cosa il client stia effettivamente richiedendo. La SSE consente di risolvere il seguente problema: dato un insieme di documenti e delle parole rilevanti, che fungono da chiavi di ricerca, è necessario identificare quali documenti contengano una determinata chiave. Ad esempio, la medicina personalizzata, che adatta pratiche mediche e farmaci al corredo genetico di ciascun individuo per massimizzarne l'efficacia, è un campo applicativo che pone la necessità di eseguire ricerche di questo tipo su dati estremamente sensibili ma di dimensioni ingenti, imponendo l'adozione di soluzioni sufficientemente performanti. La SSE potrebbe essere impiegata, ad esempio, per identificare quali genomi appartenenti ad un gruppo di individui contengano una specifica variante di un gene, celando sia il gene ricercato che i risultati della query.

Essendo le operazioni svolte sul server remoto, è inevitabile che esso apprenda alcuni dettagli relativi alle query dalla loro esecuzione: per quantificare le informazioni che il server può ricavare, ogni protocollo di SSE definisce un preciso *profilo di leakage* che determina cosa il server apprende durante l'esecuzione del protocollo. Nella maggior parte dei casi esse non sono sufficienti a stabilire la composizione originaria del *dataset*.

La ricerca di sottostringhe è un problema più generale rispetto a quello risolto dalla SSE. Essa consiste nel ricercare le posizioni in cui una sequenza di caratteri viene rinvenuta in un testo più grande. È evidente che la ricerca di sottostringhe può essere usata per la ricerca per parole chiave: infatti, è sufficiente restituire come risultati della computazione tutti i testi in cui una determinata parola occorra almeno una volta. Tecniche puramente crittografiche categorizzabili come Substring-SSE sono state impiegate per risolvere questo problema [7], ma i protocolli che ne derivano consentono ad un attaccante di capire quanto sia lungo il prefisso comune a due sottostringhe che vengono ricercate. Tale informazione può essere ricavata confrontando fino a che punto le porzioni del dataset che esse richiedono durante l'elaborazione, definite *pattern di ricerca*, coincidono. Pertanto, questo problema richiede l'adozione di tecniche finalizzate a celare completamente il pattern di ricerca, che altrimenti rivela la similirità nei contenuti ricercati dal client.

Per tale ragione, una primitiva crittografica nota come Oblivious RAM (ORAM) viene sfruttata in [25]. Le ORAM infatti consentono di svincolare totalmente il pattern di accesso alla memoria dall'elemento al quale si è interessati, anche quando esso viene prelevato ripetutamente: esse ricorrono a continue ri-cifrature e rimescolamento del contenuto della memoria per impedire al server di associare gli accessi alla memoria ad una determinata porzione dei dati. Mentre il server memorizza la maggioranza dei dati, il client ne estrae di volta in volta una porzione, per accedere all'elemento di interesse e per effettuare le operazioni di rimescolamento. Al termine del processo, il client aggiorna il contenuto della struttura dati remota senza che il server possa capire l'elemento a cui è stato fatto accesso e come i dati sono stati riorganizzati. Il leakage profile di tale applicazione consta della sola lunghezza della stringa originale e lunghezza della query. Tuttavia, dato che parte del protocollo delle ORAM viene eseguito dal client, è necessario trasmettere una notevole quantità di dati tra client e server attraverso la rete, provocando un notevole calo di performance. Inoltre, il server ha solo la funzione di una memoria di dimensione considerevole, che non può operare direttamente sul contenuto di una ORAM essendo esso processato in chiaro dal client.

In questo contesto, le *enclavi* rappresentano una soluzione a tali problemi: infatti esse consentono di definire porzioni di memoria alle quali nessuna componente hardware o software facente parte del sistema può accedere, incluso il sistema operativo (SO) e, di conseguenza, qualunque attaccante avente i privilegi di *root*. Questo fa sì che neanche il cloud provider, che è il legittimo possessore della macchina, possa accedere al contenuto incapsulato da una enclave, che pertanto può operare su dati in chiaro senza il rischio che essi vengano rivelati. Questo consente di adottare algoritmi convenzionali senza applicarvi alcuna modifica e senza ricorrere a complesse costruzioni crittografiche, che comportano un aumento dei tempi di elaborazione. In particolare, una enclave potrebbe accedere ai dati cifrati contenuti nella porzione di memoria non protetta e decriptarli nella propria, in modo da poter operare su di essi in chiaro. Intel Software Guard Extensions (Intel

SGX) è una tecnologia integrata in tutte le CPU prodotte da Intel a partire dalla microarchitettura *Skylake* che consente di istanziare enclavi. Sebbene inizialmente ideata per impedire ai fruitori di contenuti digitali di violarne i diritti d'autore, ovvero per il Digital Rights Management (DRM), Intel SGX si adatta bene anche allo scenario del cloud computing, e in particolare alla possibilità di processare informazioni confidenziali. Oltre a garantire confidenzialità, le enclavi di Intel SGX sono sottoposte ad un processo di *attestazione remota*, che ne garantisce l'autenticità. Infatti, il possessore della macchina potrebbe eseguire un'enclave differente da quella prevista dal client che ne fa uso: pertanto ricorrere all'attestazione remota consente di verificare l'autenticità dell'enclave e garantire che il possessore della macchina non abbia barato. Una enclave speciale, che ha l'accesso esclusivo a chiavi di attestazione integrate sul *die* della CPU, genera una prova crittografica che può essere verificata accedendo ai servizi di attestazione esposti da Intel, ovvero Intel Attestation Service (IAS).

Tuttavia, l'assunzione che un avversario non abbia la possibilità di estrarre segreti durante l'esecuzione di un'enclave ha alimentato l'interesse nei confronti dei reali margini di sicurezza garantiti dalle enclavi, e quali attacchi sia possibile sferrare per comprometterli. In particolare, le Intel SGX, da specifica, non implementano alcuna contromisura contro gli attachi *side channel*, che mettono a rischio la confidenzialità sfruttando sorgenti di informazione collaterali, dovute alla maniera in cui tale tecnologia è implementata. In particolare, un avversario a livello di root è in grado di ispezionare la sequenza di indirizzi virtuali a cui un'enclave accede, ovvero il suo *pattern* di accesso alla memoria. Dato che i programmi usualmente effettuano salti condizionali o accedono a indici di un array che dipendono da variabili contenenti un segreto, l'analisi degli indirizzi a cui un'enclave effettua l'accesso è estremamente efficace nell'inferire tali segreti. La granularità degli indirizzi che un attaccante può discriminare è di una *pagina*, che ha dimensione di 4 kB su architetture `x86_64`. Tuttavia, essa è sufficiente a estrapolare un'ingente quantità di informazioni, ad esempio i contorni di una immagine in formato JPEG processata all'interno di un'enclave [42]. Tali attacchi, definiti *page table attacks* per via della loro risoluzione, sono deterministici e in grado di estrarre segreti durante una singola esecuzione dell'enclave. Attacchi più complessi, i cosiddetti *cache attack*, sfruttano il fatto che la *cache L1* è condivisa tra i processori logici quando l'*hyperthreading* è attivo, consentendo ad un thread attaccante di interferire con gli accessi a memoria effettuati dall'enclave quando essi sono in esecuzione sullo stesso processore fisico. Sebbene tali attachi abbiano una granularità più fine, richiedono esecuzioni multiple dell'enclave per via del fatto che non sono deterministici [2].

Gli attachi presenti nello stato dell'arte mostrano come sia possibile ricostruire il contenuto di chiavi cifrate o i dati processati da librerie preesistenti quando esse vengono integrate in una enclave senza apportarne modifiche. Vi sono diverse contromisure atte a garantire la retrocompatibilità di software esistente, alcune basate su memoria transazionale [20] [34], altre basate su modifiche dei programmi eseguite all'atto della compilazione [35]. Il loro svantaggio è che oltre ad introdurre un significativo rallentamento, in genere risolvono solo i problemi relativi ad un unico side channel. Dato che la combinazione di diverse contromisure potrebbe essere difficile da implementare, è evidente che sia necessario un approccio più solido che rappresenti una soluzione di lungo termine.

In particolare, le problematiche che sono state esposte possono essere risolte implementando algoritmi che siano *oblivious*, ovvero tali che la sequenza di operazioni che essi eseguono non dipenda dal loro input. Ciò comporta la realizzazione di algoritmi che evitino sia di eseguire salti condizionali (mantenendo un flusso di esecuzione a tempo costante) sia di accedere a strutture dati con indici dipendenti da valori segreti. Quest'ultimo requisito è soddisfatto dalle ORAM, che infatti rendono il pattern di accesso alla memoria e l'elemento richiesto completamente scorrelati anche a seguito di accesi ripetuti. Nonostante i protocolli di ORAM tradizionali prevedano uno scenario remoto, in cui solo il client ha accesso alle informazioni in chiaro, l'uso di enclavi consente concettualmente di spostare il client su un'enclave in esecuzione sul server. Tale approccio ha due evidenti vantaggi: innanzitutto, non è richiesto che il client implementi in locale l'algoritmo, dato che anche esso può essere eseguito in enclave; in secondo luogo, non è necessario scambiare dati attraverso la rete, essendo i trasferimenti tra client e server eseguiti nell'ambito della stessa macchina. Il client dell'enclave svolge le operazioni fondamentali per la sicurezza, in particolare il rimescolamento del contenuto della ORAM. Bisogna tuttavia tenere in considerazione che tali operazioni sono esposte ai side channel: un avversario in grado di scoprire come essi vengono ridisposti vanificherebbe l'uso delle ORAM. Per fronteggiare questo problema, i protocolli ORAM *doubly oblivious* [31] [24] implementano un client che non espone alcuna informazione all'esterno.

La ricerca di sottostringhe che opera su testi di dimensione considerevole (nell'ordine delle centinaia di MB o qualche GB) richiede la creazione di strutture di indicizzazione che consentono di effettuare una query in tempo sublineare rispetto alla lunghezza del testo. Tuttavia, gli indici non solo sono molto più onerosi da memorizzare rispetto al testo originale, ma richiedono una serie di accessi casuali, che potrebbero rivelare facilmente la similarità delle sottostringhe ricercate. Per risolvere il problema dell'occupazione di memoria e processing si può ricorrere ad una enclave istanziata su un server remoto, mentre per proteggere il pattern di ricerca, gli indici possono essere memorizzati in una struttura dati basata su ORAM. Oblivious Substring Query on Remote Enclave (ObSQRE) è un innovativo protocollo di ricerca sicura di sottostringhe, che integra diversi indici testuali per effettuare query in modo efficiente. In particolare, impiega tre differenti indici, specificando per ciascuno di essi un modo opportuno per proteggerne il contenuto mediante delle ORAM. I protocolli di ORAM che adotta sono tre, ovvero la Path, Circuit e Ring ORAM. Oltre a fornire una implementazione doubly oblivious per la Path e la Circuit ORAM, ObSQRE estende tale approccio anche alla Ring ORAM, della quale non esiste una versione doubly oblivious nello stato dell'arte. Nessuna di queste primitive è stata impiegata per risolvere il problema della ricerca di sottostringhe sfruttando l'esecuzione in enclave.

I tre indici che vengono impiegati derivano dalla Burrows-Wheeler Transform (BWT) del testo e dal suo *suffix array*, e consentono di implementare le rispettive versioni dell'algoritmo di *backwards search*, che processa i caratteri della sottostringa da ricercare a partire dall'ultimo. Abbiamo modificato gli algoritmi in modo da tenere in considerazione i requisiti di sicurezza e le vulnerabilità delle enclavi: in particolare, sono stati eliminati tutti i salti condizionali e gli accessi agli indici sono mediati dalle ORAM, garantendo che l'esecuzione di ogni ricerca sia totalmente oblivious, e pertanto, resistente ad ogni tipo di attacco side channel che si basi sull'ispezione del pattern di accesso alla memoria. Le soluzioni finali rivelano solo la lunghezza iniziale del testo e il numero di caratteri dell'alfabeto che compone le stringhe, ed è possibile anche offuscare la lunghezza della query nonché il numero di risultati. Se n è la lunghezza del testo ed m il numero di caratteri della query, le soluzioni che presentiamo eseguono la ricerca in $\mathcal{O}(m \cdot \log^2 n)$. Nonostante la complessità sia asintoticamente la stessa per tutti gli algoritmi, le costanti moltiplicative sono abbastanza rilevanti da identificare una soluzione migliore delle altre in tutti i test. Per validare il nostro approccio, consideriamo testi di lunghezza variabile basati su alfabeti di diversa dimensione, quali i caratteri ASCII, le basi azotate che compongono il DNA e gli amminoacidi che costituiscono le proteine. Sebbene alcune soluzioni mostrino variazioni delle perfomance a seconda dell'alfabeto considerato, un indice in particolare ottiene gli stessi risultati a prescindere da esso. I test dimostrano che le costruzioni basate su Circuit ORAM sono le più efficienti salvo per alcuni casi, in cui, tuttavia, lo scarto di performance è trascurabile. L'algoritmo migliore consente di eseguire una query di 3050 caratteri, su un testo di 32 MB in 537 ms, ottenendo un tempo di esecuzione di circa 176μs per ogni carattere di sottostringa. Ciò rende ObSQRE un protocollo interessante anche in scenari applicativi reali, dato che sottostringhe di tale lunghezza codificano un gene all'interno del DNA.

Un ulteriore contributo consiste nell'attenta analisi dell'attestazione remota. Infatti Intel l'ha ideata prevalentemente per applicare tecniche di DRM, che consentono di evitare la propagazione e l'abuso di materiale protetto dal diritto di autore. Nello scenario da loro prospettato, un client che richiede l'accesso a contenuti protetti effettua una richiesta ad un server, che li invia in forma cifrata ad una enclave solo dopo che essa si sia autenticata. In tal merito, il provider di tali contenuti coincide con l'autore o sviluppatore dell'enclave, o Independent software vendor (ISV). La procedura standard prevista da Intel SGX consente solo all'ISV di attuare l'attestazione remota ed accedere all'apposito servizio di Intel, l'IAS. Nel nostro scenario, un client che non necessariamente è l'autore di una enclave, vuole eseguire software sicuro su un server potenzialmente malevolo. Dettagli implementativi del protocollo di attestazione normalmente impedirebbero di eseguirla e richiedere all'IAS il responso sull'autenticità dell'enclave. Tuttavia, applicando modifiche minimali che non intaccano la sicurezza complessiva, è possibile adattare il protocollo originario anche a seguito di un cambio dei suoi requisiti. Tutti i lavori che usano le Intel SGX danno per scontata questa possibilità o trascurano tale problema. Pertanto, siamo stati i primi a fornire una analisi puntuale della sicurezza del protocollo nello scenario del cloud computing e a proporre una maniera per poterlo mettere in atto.

In sintesi, il nostro lavoro fornisce i seguenti contributi.

- Design ed implementazione di ObSQRE, un protocollo efficiente per la ricerca di sottostringhe in uno scenario remoto garantendo elevati livelli di confidenzialità, che includono la protezione del testo, del contenuto delle sottostringa cercata, e del risultato della query, impedendo all'attaccante di correlare sottostringhe cercate in query distinte. ObSQRE combina l'esecuzione in enclave tramite Intel Software Guard Extensions (Intel SGX) e le doubly oblivious ORAM per la realizzazione di indici testuali oblivious, che consentono di effettuare ricerche in tempo sublineare rispetto alla lunghezza del testo, ottenendo risultati di interesse pratico.

- Design ed implementazione di una versione doubly oblivious della Ring ORAM, mai presentata in letteratura, con ottimizzazioni che tengono in considerazione lo scenario in cui il client della ORAM non è remoto, ma ha accesso diretto ai dati contenuti nella ORAM.

- Esecuzione di test estensivi per valutare le performance di diversi protocolli ORAM doubly oblivious, in particolare Path, Ring e Circuit ORAM. I risultati sono confrontati con quelli delle controparti non doubly oblivious, per stimare l'impatto che un client oblivious ha sulle performance.

- Analisi del protocollo di attestazione remota di Intel finalizzata alla sua adozione nel contesto di outsourcing della computazione nel cloud, superando i limiti della procedura di Intel che ne ostacolerebbero l'adozione nello scenario considerato nel nostro lavoro.

Contents

Acronyms

AE Authenticated Encryption. 67, 70, 71, 94, 114

AES Advanced Encryption Standard. 19, 26, 27, 94, 110, 121, 133

AES-NI AES New Instructions. 133

AEX Asynchronous Exit. 20

BWT Burrows-Wheeler Transform. 8, 61, 62, 63, 66, 97, 98, 99, 100, 114, 133, 137, 142, 143, 144, 153, 140, 154, 157, 147

CLI command line interface. 135

CMAC Cipher-based Message Authentication Code (MAC). 121, 123, 124

CPA Chosen-Plaintext Attack. 19

CPRNG Cryptographically Secure Pseudorandom Number Generator. 110, 111

CTR Mode Counter Mode. 37, 96, 110, 133

DHKE Diffie-Hellman Key Exchange. 117, 118, 120, 121, 124

DoS Denial of Service. 81

DRM Digital Rights Management. 17, 117, 122, 125, 162

EPC Enclave Page Cache. 13, 16, 20, 24, 71, 119, 127, 128, 133

EPCM Enclave Page Cache Metadata. 13, 119

EPID Enhanced Privacy ID. 119, 121

FHE Fully Homomorphic Encryption. 1, 2, 77

GCM Galois Counter Mode. 94, 133

HT Hyper-Threading. 24

HTTP Hypertext Transfer Protocol. 76, 82, 123, 129, 130

IAS Intel Attestation Service. 12, 117, 118, 119, 120, 121, 123, 126, 131, 162

Intel IPP Intel Integrated Performance Primitives. 133

Intel AE Intel Architectural Enclave. 72, 126

Intel ME Intel Management Engine. 11

Intel SGX Intel Software Guard Extensions. xii, 5, 6, 8, 9, 11, 12, 13, 16, 18, 19, 20, 24, 28, 66, 71, 77, 78, 81, 114, 117, 118, 119, 120, 122, 125, 127, 129, 131, 133, 137, 145, 152, 154, 157, 161, 162

ISV Independent software vendor. 12, 117, 118, 119, 120, 121, 122, 121, 122, 123, 124, 125, 126, 131, 162

IV Initialization Vector. 37, 67, 133

JPEG Joint Photographic Experts Group. 20, 22

JSON JavaScript Object Notation. 130

KDK Key Derivation Key. 121, 122, 123

LCP Longest Common Prefix. 59

MAC Message Authentication Code. xvii, 72, 94, 121, 122, 123, 133

MITM Man-in-the-middle. 121, 122, 124, 125, 131

MK Master Key. 123

MMU Memory Management Unit. 14, 16

ObSQRE Oblivious Substring Query on Remote Enclave. i, iii, xxii, 8, 9, 81, 82, 114, 115, 118, 129, 137, 154, 157, 165

ODS Oblivious Data Structure. 51, 73, 81, 101, 102, 105, 107, 110, 112, 113, 114, 115, 126, 132, 133, 153, 154, 162

ORAM Oblivious RAM. xxi, xxii, 3, 5, 7, 8, 9, 11, 29, 30, 32, 33, 36, 37, 38, 39, 42, 44, 47, 48, 49, 50, 51, 66, 70, 73, 74, 77, 78, 79, 80, 81, 76, 82, 83, 85, 88, 89, 91, 93, 94, 96, 97, 99, 100, 101, 107, 110, 113, 114, 115, 116, 126, 128, 129, 132, 133, 135, 137, 138, 139, 140, 142, 143, 144, 145, 149, 151, 153, 154, 157, 161, 162, 163

OS operating system. 5, 11, 12, 14, 15, 16, 18, 19, 20, 23, 24, 25, 27, 129

PBKDF2 Password-based Key Derivation Function 2. 133

PMH Page Miss Handler. 16, 17, 24, 25, 27

PRF Pseudo-Random Function. 67, 74

PRM Processor Reserved Memory. 13, 16, 137

PRP Pseudo-Random Permutation. 36, 37, 38, 39, 88

PSW Platform Software. 12, 137

QE Quoting Enclave. 28, 118, 121, 123, 125

RAM Random Access Memory. 28

RSA Rivest-Shamir-Adleman Cryptoscheme. 19, 20, 26, 27, 119

SHA-2 Secure Hash Algorithm 2. 119, 123, 125

SigRL Signature Revocation List. 119, 122, 121

SK Session Key. 123, 126, 131, 135

SLA Service-level agreement. 1

SMC Secure Multiparty Computation. 1, 2

SMK Sigma Key. 122, 123

SO sistema operativo. vii

SPID Service Provider ID. 120, 122

SSE Symmetric Searchable Encryption. 2, 3, 5, 66, 74, 77

STBWT Search-tree BWT. 100, 101, 102, 105, 107, 110, 113, 115, 133, 153, 140, 154, 153, 154, 147

TCB Trusted Computing Base. 5, 12, 77, 125, 129

TLB Translation Lookaside Buffer. 16, 24, 25

TSX Intel Transactional Synchronization Extensions. 7, 23, 26, 27, 28

VK Verification Key. 123

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The current trend when dealing with huge amount of data is outsourcing them to cloud storage rather than keeping them on local machines. In fact cloud storage providers guarantee high Service-level agreements (SLAs), making data available at any time and with relatively low latency, but also shared and synchronized among all the users that have proper access permissions, since the remote machines where data is kept are accessible through the network. In order to achieve the same degree of availability, it's necessary to arrange a private cluster with huge backing storage, large network bandwidth to serve multiple requests and very high reliability, which usually comes at the cost of replication of both servers and storage devices, in order to face potential data losses due to disk failures. Besides the price of the hardware, it is also necessary to take into account the costs to keep the infrastructure online, that include power and network bills, as well as the effort required for maintainance. All these factors make the employment of a private cluster unfeasible in most of the cases, especially for companies and users who don't have the technical background or budget to set up such an infrastructure. Hence, these functions are usually delegated to an external provider that offers all these services as part of her fees.

Nonetheless, outsourcing data poses the problem of confidentiality: in fact, an untrustworthy service provider may exploit sensitive information to illicitly gain benefits, potentially harming the owner. Data encryption is not a viable solution in all the cases: indeed, the outsourced data may not be composed of static files, but also be part of a huge data structure that doesn't fit the client memory but is required to perform some kind of computation. In this scenario, it would be reasonable to offload the execution of the algorithm to the cloud as well, in order to cut the necessity for the network round trips required to fetch the needed data. However, standard data encryption techniques would prevent the remote machine from performing any computation over the uploaded dataset. If performance is not a major concern, a client may decide to run the algorithm locally, fetching portions of her encrypted dataset and operating on plaintext only on her own machine to protect against eavesdroppers. Even in this case, if the cloud service provider knows which kind of processing is carried out by the client, she can infer a huge amount of information by inspecting the sequence of requests issued by the client. For example, if the user uploads a data structure that returns the results that match a certain kind of search criterion, an untrustworthy server may infer the similarity of different queries by comparing the sequence of accesses during their execution, i.e. their *search pattern*. Several works showcase that with enough domain knowledge, an untrustworthy server may reconstruct substantial portions of an encrypted dataset only leveraging its knowledge of the search patterns [5,19,29]. There exist cryptographic primitives, namely Fully Homomorphic Encryption (FHE) and Secure Multiparty Computation (SMC), that allow to perform computations over encrypted data as well as reducing this information leakage. Nonetheless, they introduce a releavant overhead in the computations, up to the point of making them impractical in real world scenarios.

The problem of querying huge amounts of confidential data recurs in several situations. For example, personalized medicine adapts medical practices and drugs based on the genome of an individual, in order to maximize the effectiveness of medical treatments as well as diagnosing rare pathologies. It requires to perform thorough analysis over the genome of a patient, which is extremely sensitive since it contains most of his clinical information. Another application scenario may involve a company owning a huge amount of confidential data, such as logfiles or emails of its employees, that wants to outsource them while preserving the possibility to search for specific contents.

Symmetric Searchable Encryption (SSE) partially answers to the need for secure and efficient ways to query encrypted data. In particular, given a set of documents and a set of keywords that may be looked for, SSE solves the problem of finding the subset of documents that contain a certain keyword. Unlike FHE and SMC, SSE achieves performance levels that make it a viable solution in real world applications. SSE protocols rely on purely cryptographic constructions as well, and comply to a provable security level that is defined by their *leakage model*. It must be taken into account that some information necessarily leaks due to the fact that it's a remote party that actually carries out the computation. However, the leakage model allows to estimate the amount of information that an untrusted party learns about the original dataset or the queries while executing the protocol. A client opts for a certain solution according to the security level she wants to achieve, the trust he puts on the remote server as well as the desired security/performance tradeoff, since higher confidentiality guarantees usually come at the cost of a higher overhead. In the scenarios we envision, SSE may be adopted to find which genomes contain a sequence of nucleotides that encode specific traits, warnings or errors in logfiles or releavant keywords in emails.

The problem solved by SSE is very specific and may exhibit some limitations, such as the impossibility to add a new keyword to the remote dataset once it has been uploaded. On the other hand, substring search is a more general problem, that amounts to finding the positions of the repetitions (or *occurrences*) of a sequence of characters within a larger text. It is straightforward to notice that substring search is a generalization of the problem solved by SSE: in fact, given a set of documents, it is possible to answer a SSE query by returning the identifiers of the documents where a certain keyword occurs at least once. The solution to substring search problem also exhibits much more flexibility since it is possible to look for patterns that were not initially included among the keywords, without perfoming any update on the remote data. Indeed, a Substring-SSE based solution to the substring search problem based on suffix trees is described in [7]. While their approach allows to totally conceal the structure and content of the initial text, besides its length and the size of the alphabet, it leaks the search pattern of its queries, allowing to understand to which extent the prefixes of two searched strings match.

The leakage of the search pattern is a major concern when performing substring search, since it allows to establish the similarity between two queries just by inspecting which elements of the remote data structures are accessed. In this regard, other works aim at totally concealing the search pattern to the remote server, so that it is impossible to infer meaningful information about the queries as well. ORAMs were successfully employed to achieve this purpose [25]. ORAMs are cryptographic primitives that allow a client to fetch portions of her remote dataset without letting the server know which element was actually needed. Their property is that the server does not learn anything about the real arrangement of the data even when repeatedly accessing the same element: in fact, they resort to continuous shuffle and re-encryptions that scramble the data in such a way that the sequence of memory accesses needed to fetch a portion of the data is totally random looking and uncorrelated to the logical identifier of the requested element. The server that stores the bulk of the data does not need to have huge computational power, as the shuffles and encryptions are executed by the client after each access to the remote dataset. Hence, ORAM protocols require the constant interaction between the client and the server and usually incur a high bandwidth blowup, since they require the transfer of much more data to conceal which portion of the original dataset was requested. Since network latency is much greater than the time the client spends in the computation, it becomes the figure of merit to establish the efficiency of remote ORAM protocols. In [25], several ORAMs are used to wrap two efficient text indices, i.e. the suffix tree and array, in order to hide the search pattern of the queries as well as the content of the original text. While this protocol leaks very little information about the initial data, it only addresses the scenario in which the dataset is composed of several short strings (i.e. made of at most 100 characters) concatenated together. Even if this solution may be adapted to query slightly larger texts, it does not apply when handling texts whose size ranges from hundreds of MB to GB, such as genomes. Furthermore, the protocol requires several roundtrips for every query character, making its performance strictly dependent on the latency of the network. It is worth to notice that the client plays a fundamental role, both in ORAM management and computation. In fact, ORAMs only provide a way to conceal the search pattern of an application, but don't enable the execution of operations over encrypted data: the algorithm to query the remote data structure needs to operate on plaintext data and is totally implemented on the client side.

Hence, this approach in general provides stronger security guarantees but fails in allowing to offload the computation to the remote server.

All the solutions to the substring search problem that solely rely on cryptographic constructions exhibit strong limitations: while Substring-SSE carries out the computation on the server, it incurs an information leakage that may be significant and harmful in extremely sensitive contexts, such as when handling biological data; on the other hand, solutions that rely on ORAMs require a lot of interactions between the client and the server and lose the amenable possibility to offload the computation to a remote party.

If the owner of the remote machine was trusted, it would be possible to perform computations on plaintext data: while this situation is unlikely to happen when dealing with an unknown and possibly untrusted entity, it would be sufficient that only the portion of the machine carrying out sensitive computations was trustworthy. All the components in the Trusted Computing Base (TCB) of a system are provably secure against all the attacks that belong to the threat model they assume. The cloud computing scenario poses a tough challenge in this regard: in fact, the whole software stack running on a server is potentially compromised, including the operating system (OS) or hypervisor. Hence the threat model of cloud computing usually includes adversaries with root permissions, that have full ownership of the machine and may interfere with the computations at their will. This is where enclaves come into place: an enclave is a secure execution environment hosted within an untrusted system that provides strong guarantees of confidentiality and integrity. An enclave encapsulates code and data in such a way that neither any portion of the software, nor any hardware component may access their content: this includes the OS, hypervisor, peripherals and even other enclaves allocated on the system. Hence, an enclave allows to define a TCB even on a remote machine owned by an unfaithful cloud provider that is interested in learning confidential data. They are used to wrap and safely execute the sensitive portions of a program: since their content cannot be inspected, they can implement algorithms that operate on plaintext data without running the risk of leaking confidential information. In particular, while the bulk of the data is stored in encrypted form in the untrusted portion of the system, it is decrypted once moved inside an enclave, that can safely store the cryptographic keys of the dataset as well.

Intel SGX is one of the few commercially available implementations of enclaves, as they are included in all the CPUs since the *Skylake* microarchitecture. Besides the possibility to allocate private memory regions that contain the sensitive portions of a program, it also offers the possibility to perform *remote attestation*, a procedure that provides a strong cryptographic proof of the authenticity of an enclave: it attests that the enclave has been instantiated with the expected content, preventing the server from replacing it with a rogue one that implements malicious behaviors. However, the remote attestation protocol devised by Intel is mainly conceived for the Digital Rights Management (DRM) scenario, allowing a provider of copyrighted contents to protect them from a client potentially interested in infringing the terms of use. In this setting the content provider, that corresponds to the enclave developer, delivers and processes such protected data using an enclave of its own, after ensuring its authenticity. Incidentally, Intel is involved directly in the remote attestation process, and only grants access to this service to enclave developers. Conversely, in our scenario, a client is willing to attest an enclave, whose author is a third party developer, and that is allocated on a remote server. The attestation procedure of Intel does not apply straightforwardly to our scenario, thus limiting the adoption of Intel SGX based solutions to outsource computation on a remote server. To overcome these hindrances, we provide a thorough analysis of the attestation protocol showing how to adapt it to our needs. Surprisingly enough, none of the works that exploit Intel SGX in cloud computing scenario have ever provided a description of how to circumvent these limitations.

While enclaves seem to offer a technological solution that totally supersedes purely cryptographic constructions, the strong attacker model they assume has drawn the attention over their actual security. It turns out that indeed they are exposed to subtle attacks that compromise the confidentiality of the contents they are supposed to protect. In particular, they are vulnerable to side channels, that are unwanted sources of information that derive from the way that Intel SGX is integrated into existing platforms. The main source of side channel leakage of Intel SGX is the memory access pattern: in fact, a privileged attacker has the means to inspect the sequence of memory addresses accessed by an enclave, from which it can infer releavant information regarding the control flow of the application. In particular, enclaves leak significant information when they branch on conditions evaluated on secrets or access memory locations based indices that depend on confidential data. While these programming practices are totally legitimate in software whose focus is not security, they turn out to be a major concern when developing applications that need to comply to high security standards. In fact, these attacks are successful in retrieving significant portions of sensitive information when retrofitting legacy applications that don't have the strong attacker model of Intel SGX in mind [2, 3, 42].

While there are mitigations that protect legacy applications even when running within an enclave [20, 34, 35], they mostly act as stopgaps, as newer attacks that exploit subtleties and undocumented behaviours of CPUs may totally dismantle the security they try to achieve. The countermeasures are usually compiler based solutions that aim at concealing the control flow of a program or rely on Intel Transactional Synchronization Extensions (TSX) to hide the memory access pattern of the enclave. Their major limitation is that they usually address one source of side channel leakage at a time, and don't simultaneously protect from all the attacks that have been conceived. Furthermore, they usually incur a quite high performance penalty: hence, combining several mitigations may be impossibile, impractical or introduce such an high overhead that the final solution is not practical.

A rich line of works explores methodologies to build countermeasures directly into new applications, taking into account security concerns related to side channels at design time. Some works resort to oblivious algorithms [43], whose control flow and memory access pattern does not depend on input data, and thus are not prone to the inspection of the memory access pattern. While they incur some overhead with respect to their non-oblivious counterpart, they solve the problem at its root, in a way that cannot be thwarted by new side channels that allow to inspect the sequence of memory locations accessed by the enclave. Another line of works focuses on the construction of data structures that prevent the server from knowing which element is accessed [1]. Notably, an ORAM protocol whose client is executed within an enclave is presented in [24] and [31], that implement an enclaved version of Path and Circuit ORAM respectively. Executing the ORAM client on the same machine that stores the bulk of its data allows to dramaticaly improve performance, since there is no more need to transfer data over the network when accessing it. Furthermore, the data that is stored in encrypted form can be safely decrypted and processed within the enclave, making it possible to offload the whole computation on the remote server unlike traditional remote ORAM protocols. However, this requires a careful revision of the operations performed by the ORAM client, that must be implemented in such a way that the server doesn't learn how data is shuffled between subsequent accesses. In fact, data shuffles and re-encryptions are critical steps in ensuring that there is no correlation between the elements that populate the ORAM and the sequence of memory location accessed to retrieve them. ORAM protocols that conceal the operations performed on the client side, besides concealing which element of the remote dataset is accessed, are defined *doubly oblivious*.

The purpose of Oblivious Substring Query on Remote Enclave (ObSQRE) is performing substring search queries in such a way that a malicious server doesn't learn any information regarding the original text besides its length and alphabet, i.e. the set of characters that compose it, and cannot establish any correlation between the queries. We exploit enclaved execution to provide confidentiality and allow the client to offload the data as well as the computation to a remote machine. To the best of out knowledge, there are no works in the state of the art that address the problem of oblivious substring search exploiting enclaves, and in particular, Intel SGX. In order to guarantee good performance, we adopt solutions that preprocess the initial text to produce an index which can be later used to perform queries in sublinear time with respect to the length of the text. Hence, we address the case in which the given dataset is a very long string, that is frequent when dealing with biological data, such as genomes, and logfiles. Since full-text indices require to perform scattered memory access that leak the search patterns of queries, we wrap them into doubly oblivious ORAMs in order to prevent the server from inferring the content of the text and the similarity between queries. On the other hand, the substring search algorithms are implemented in such a way that they don't branch on conditions that depend sensitive data. To fulfill this purpose, we adopted an algorithm known as *backwards search*, whose regular control flow is amenable for our scenario, while it achievies the same perfomance of much more complex data structures. We implement three different versions of backwards search, exploiting full-text indices derived from the Burrows-Wheeler Transform (BWT) of the original text as well as its suffix array, and compare their performance. Our protocol allows to retrieve the number of occurrences of a substring in a single round of communication.

Our work provides several contributions, that we now summarize.

- We design and implement ObSQRE, a private substring search protocol that leverages Intel SGX and *doubly oblivious* ORAMs to build oblivious full-text indices. We implement 3 different variations of *backwards search*, allowing to perform substring search queries in sublinear time in the lenght of the text in order to achieve performance of practical interest. Our solution hides the structure of the text, the content of the substring searched and the result of the queries, as well as the similiraties among substrings searched in distinct queries.

- We provide a doubly oblivious construction of Ring ORAM, that is not present in the state of the art. In particular, we optimize it to take into account the fact that the client and the server of the ORAM reside on the same machine.

- Besides Ring ORAM, we also implement existing doubly oblivious Path and Circuit ORAMs, and we perform extensive benchmarks in order to establish their performance when coupled with Intel SGX. Furthermore, we compare them with their singly oblivious counterpart to estimate the performance overhead due to the implementation of an oblivious client.

- We review the remote attestation procedure devised by Intel in order to adapt it to the scenario of outsourced computation in the cloud. In particular, we show how a client who is not an enclave developer can access the remote attestation facilities run by Intel.

Chapter 2

State of the Art

In this chapter we introduce all the concepts that will recur throughout the work. In Section 2.1 we present Intel SGX, explaining the basic mechanisms behind its security, the threat model it assumes and the rich literature of works that leverage side channels to extract confidential information during the execution of an enclave, as well as a series of possible mitigations. In Section 2.2 we provide the details about several ORAM constructions, starting from Path ORAM, that paved the way for more elaborate primitives such as Ring and Circuit ORAM. We explain their function and how to optimize ORAMs according to the access pattern of the application that employs them, introducing the concept of Oblivious Data Structures. Section 2.3 describes existing full-text indices that provide useful background for the substring search problem in a safe setting. Finally, the related works described in Section 2.4 either show how to exploit Intel SGX enclaves to guarantee confidentiality, possibly combining them with ORAM constructions, or provide some background about purely cryptographic techniques to perform remote substring search in the outsourcing scenario.

2.1 Intel SGX

Intel Software Guard Extensions (Intel SGX) is a technology shipped with Intel CPUs since the *Skylake* microarchitecture. It allows to instantiate protected and trusted execution environments, called *enclaves*, which provide strong confidentiality and integrity guarantees for both the code and the data they encapsulate, even when running on an untrusted system. Each enclave is allocated in a portion of the physical memory that is unaccessible to all of the software and hardware composing the system (except for the CPUs) including hypervisor, OS, Intel Management Engine (Intel ME) and other enclaves [12].

The strong security guarantees provided by Intel SGX introduce a new threat model, which includes root-level adversaries and malicious cloud service providers that have full control over the attacked machine. Since the purpose of Intel SGX is to reduce the Trusted Computing Base (TCB) to the sole CPU, the final user will need to trust only the CPU hardware and its manufacturer, rather than other third-party vendors and service providers. This especially comes in hand when she needs to offload intensive computations over confidential data in the cloud, where privacy may be a major concern.

Although enclaves are set up by the untrusted OS of a remote machine, the enclave developer or Independent software vendor (ISV) can verify that the code and data they contain is legitimate. The *remote attestation* procedure allows enclaves to provide a cryptographic proof of their authenticity, named *attestation report*. Intel must be considered a trusted party in the process: in fact, the signature appended to the report is generated by a special enclave signed by Intel and can be verified by the ISV only by querying the Intel Attestation Service (IAS), which is run by Intel on its own premises. Thus, the Intel SGX ecosystem, besides a set of architectural features, includes also a whole SDK, the services to attest an enclave and the Platform Software (PSW), that includes the special enclaves needed to sign attestation reports. We now provide a brief overview of Intel SGX technology, followed by the threat model assumed by Intel and the security weaknesses which must be taken into account in the design of SGX applications.

2.1.1 Technical description

An Intel SGX based application is composed of two portions: an untrusted one which contains code running outside the enclave and data which is accessible to the host machine, and a trusted one, which is composed by one or more secure enclaves employed in the application. These enclaves are distributed as *shared objects* (`.dll` files on Windows OS and `.so` files on Unix-like OS's) that are loaded into memory using specific functions included in Intel SGX libraries. Therefore, the content of an enclave can be inspected and reverse engineered allowing an end user who is not the enclave developer to verify that the behavior implemented by an ISV is the one it advertizes. As any other library, each enclave exposes an interface, i.e. a set of functions that can be called by the untrusted code: it defines the only locations where a process can jump to start executing enclave code. This operation is performed via a specific instruction, `EENTER`, that switches the logical processor to *enclave mode*. Only a process running in user mode can call `EENTER`: since no privilege switch happens when entering enclave mode, a malicious process executing an enclave cannot take over or harm the hosting system. Furthermore, as neither the privileged system software may read or write the content of an enclave, after a successful remote attestation the end-user of the enclave can trust its code.

An enclave can be conceived as a portion of memory that is accessible only by the code it encapsulates. Therefore, part of the main memory must be reserved for enclaves that may be allocated on the system. The Processor Reserved Memory (PRM) serves this purpose. Its size of 128MB stores both enclaves and metadata: the former reside in the Enclave Page Cache (EPC), which is about 96MB, while the latter in the Enclave Page Cache Metadata (EPCM). The EPCM tracks which parts of the EPC are valid and the enclave owning them.

It is responsibility of the CPU to protect the PRM from rogue memory accesses: when an unauthorized process tries to access the EPC, the operation triggers the *abort transaction semantics*, that skips write operations and returns the integer value -1 on read operations. On the other hand, any attempt to tamper with the EPCM triggers a *page fault*: only Intel SGX dedicated instruction can manipulate data contained in the EPCM. External peripherals cannot issue *DMA transactions* targeting the PRM, since the *memory controller* residing in the CPU ignores them. Moreover, whenever data that will be written to the PRM leaves the CPU package, it is transparently encrypted, so that any attempt to directly leak secrets from the data bus (a technique known as *tapping*) fails.

The strict memory access policy required by Intel SGX must not interfere with the normal functioning of a system: one of the design goals was to implement this technology with little modifications to existing hardware and computational model, both to ease its adoption and to avoid drastic modifications to system software. The memory protection mechanism adopted by Intel SGX is tightly coupled with the *address translation* logic of modern CPUs [12]: we briefly review the memory management techniques employed in modern system software, and supported by `x86_64` CPUs, to later describe the specific modifications introduced by Intel SGX.

In current OS's, each process running on a machine lives in a separate *virtual address space*, whose size is usually much bigger that the physical memory available on the system (by contrast, this is usually referred to as *physical address space*). This abstraction gives each process the illusion of running on an independent system equipped with enough memory to cover the whole range of addresses it may access. The OS is in charge of mapping virtual addresses to the physical ones that are used to fetch data from the main memory. This operation, known as address translation, would be costly if implemented without hardware support. This is why all current general purpose CPUs include a Memory Management Unit (MMU) which is involved in the whole process.

This memory management technique carries along many advantages. The most obvious one is that executables are *relocatable*, i.e. agnostic of the physical memory addresses they will be loaded to. In fact, the addresses they access no longer refer to physical memory: this means that two distinct processes can reference overlapping virtual address ranges without contending the same memory locations, given that the OS maps them to different portions of the physical address space. Moreover, address translation prevents a process from reading and writing memory locations owned by another process, a desirable security and safety property known as *memory isolation*.

The virtual address space is partitioned in fixed-size units named *pages*. Memory is granted to processes at the granularity of pages, in order to leverage locality and limit the negative effects of memory *fragmentation*. Address translation maps pages to memory *frames*, which are ranges of the physical address space with the same size of a page. From now on, the term page will be used to refer to an address range in the virtual address space of a process, while a frame refers to an equally large portion of the physical memory installed on the system. The page-frame mapping may change during the lifetime of a process, and in extreme cases, when the system runs out of physical memory, the least-frequently used pages may be *swapped out* on a larger backing storage device. This comes at the cost of a severe time penalty when accessing a swapped-out page, as it needs to be moved to a free frame from a much slower device such as an hard-drive.

A virtual address is divided roughly in two parts. The least significant bits are used as a page offset, and thus address translation does not modify them. The remaining part, the virtual page number, is translated to a physical frame number. Page size is required to be a power of two in order to ease the retrieval of page offset and page number. In `x86_64`, the virtual address space is 48-bits wide, while the size of each page amounts to 4kB: thus, the page offset is 12-bits wide, to index each of the 4kB in a page, while the virtual page number is composed by the remaining 36-bits of the address. The virtual addresses are stored in sign-extended representation, as they are shorter than the 64-bits wide words of the CPU. While the maximum size of the physical address space is 52-bits, the current platforms only support shorter physical addresses, according to the memory requirements of different market segments. Desktop CPUs support 40-bits physical addresses, while server platforms 44-bits ones.

In order to perform the mapping, it is necessary to traverse an OS-managed data structure called *page table*, by using the different fields composing a virtual address (as in Figure 2.1). Due to its size, the page table is organized in a hierarchical way as a 512-ary tree, whose root address is stored in the special register `CR3`. It points to the so called Page Map Level 4 (PML4), which contains 512 entries. The most significant 9 bits of the virtual page number represent the index of the table where the pointer to a Page-Directory-Pointer Table (PDPT) is stored. The PDPT is another table with 512 entries that is indexed with the following 9 bits of the virtual page number to retrieve the Page Directory (PD). In an analogous way, a fragment of the Page Table (PT) is retrieved from the PD: indexing the PT yields the physical frame number associated to the initial virtual page, thus completing the *page table walk*. Each entry in these tables is 8 bytes wide: therefore, the size of each table is $512 \cdot 8\,\mathrm{B} = 4096\,\mathrm{B} = 4\,\mathrm{kB}$, exactly fitting a memory frame. The entries accomodate both a 40-bit physical frame number and several flags that are used by the OS for memory management. Among these, the *accessed bit* is set whenever a particular page is accessed,

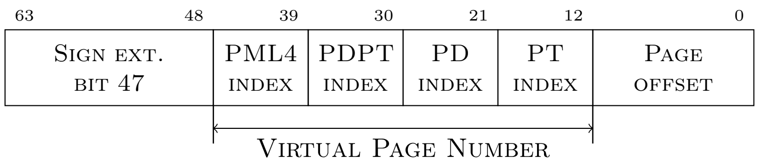

Figure 2.1: Representation of the fields composing an `x86_64` virtual address

the *dirty bit* is set on write operations and several *protection bits* impose read, write and execute restrictions to the pages. In case there is no physical frame mapped to the virtual page, or the *present bit* is not set, then a *page fault* is triggered, which requires the OS to reserve a new physical frame and to update the page table with the mapping between the requested virtual page number and the physical frame.

A page table walk is an expensive operation, that requires four memory accesses just to retrieve a physical address. In such cases, caching is beneficial since it avoids frequent retranslations of the same address, that would occur due to the *locality* of memory accesses: a special cache, the Translation Lookaside Buffer (TLB), holds the mapping between virtual page and physical frame numbers. In order to speed-up page table walks that occur with TLB-misses, the MMU resorts to specialized hardware, the Page Miss Handler (PMH). The PMH is in charge of executing all the required checks before granting access to memory. Thus, the address translation is entirely performed in hardware by the TLB and PMH; the OS is invoked only in case a page fault is detected by the PMH.

In an Intel SGX based application, enclaves are loaded at runtime in the virtual address space of the calling process. This area is named `ELRANGE` (enclave linear address range) and is mapped to frames belonging to the EPC. After completing address translation, Intel SGX embeds additional checks in the control flow of the PMH in order to prevent any code outside the enclave from accessing memory owned by the enclave itself. As outlined in Figure 2.2, access to a memory address which belongs to `ELRANGE` is granted if and only if the logical processor is in enclave mode and the corresponding physical address belongs to an EPC frame owned by the current enclave, while in case the virtual address is not in `ELRANGE`, the PMH needs to check that the address is not mapped to the PRM. In case of success, the result

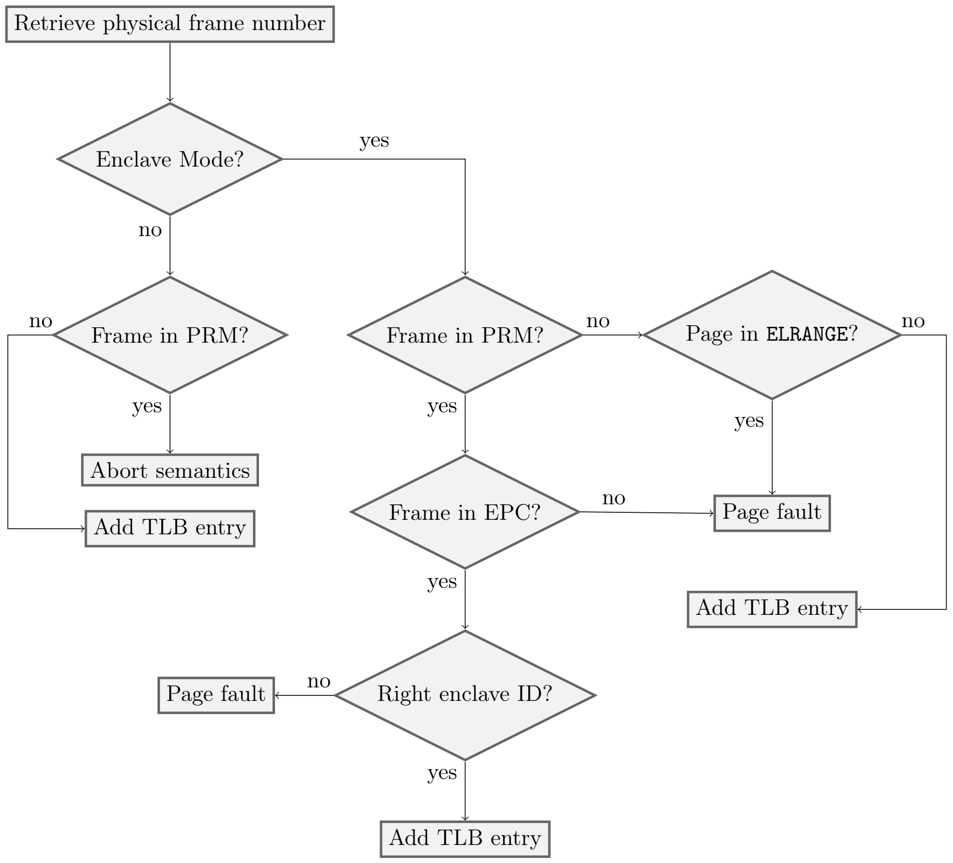

Figure 2.2: Simplified view of the security checks performed by the PMH

of address translation is added to the TLB: when this happens, further references to that memory locations are considered legit and the CPU doesn't apply any security check. Hence, whenever the logical processor exits enclave mode, either after returning from a function or due an exception, the TLB is flushed to prevent subsequent unchecked accesses to enclave memory. All the machine instructions introduced to support enclave creation, execution and attestation are implemented in *microcode*, and can thus be updated to fix security vulnerabilities that may arise due to the complexity of the checks we have just described.

2.1.2 Threat Model and Side-Channel Attacks

The main purpose of secure enclaves is allowing to instantiate a secure and trusted execution environment within an untrusted system. This necessity arises in several contexts where a computation needs to avoid the leakage of protected or sensitive data. One notable case is Digital Rights Management (DRM), that attempts to prevent illicit distribution, abuse or license bypassing of copyrighted works. For example, streaming services are likely to hinge upon secure enclaves to prevent replay of digital contents. Another scenario involves cloud services, that offer the possibility to rent computing resources in situations where the cost to buy and maintain a private infrastructure would be much higher. The use of enclaves doesn't require the cloud providers to be trusted, since sensitive data can be exchanged with the external world in encrypted form.

In the cloud computing scenario, the whole software stack running on the machines is considered untrusted: in fact, a cloud provider may employ compromised hypervisors or host OS's that extract sensitive information for profit. Therefore, the threat model of Intel SGX considers root-level adversaries with full control over the machine. Unlike other technologies, hardware attacks that compromise the CPU package are not included in the threat model [12], but they are mitigated using per-CPU keys which are concealed on the die using special circuitry. In this way, a costly imaging attack to reverse-engineer the CPU allows to compromise the keys of one or a few processors, thus making large scale attacks extremely expensive. Moreover, the threat model does not include *address bus tapping*: indeed, while memory encryption, originally meant to enforce confidentiality, has the side effect of thwarting *data bus tapping*, the address bus is totally exposed. Nonetheless, besides the costs of bus tapping attacks, the information that can be retrieved by inspecting the memory address bus is too incomplete to mount a successful attack [12]. This is mainly due to caching, which avoids bus transactions when frequently accessing the same memory locations: since many data are cached due to the temporal and spatial localities of the applications, the information which can be retrieved is too sparse and noisy to leak some valuable knowledge. Active side channel attacks that induce faults over the memory buses with the purpose of tampering with encrypted data are not considered as a part of the threat model as well.

In section 2.1.1 we highlighted that enclaves are distributed as shared objects, which are not encrypted and can thus be inspected. The remote attestation procedure allows to verify that the code and data contained in these shared objects are correctly loaded into the enclave without being tampered by the untrusted machine. Obviously, it must be ensured that pieces of code which are not verified during the attestation are not run inside the enclave, as this would allow the adversary to easily learn the data sealed inside the enclave. This necessary requirement implies that the code inside an enclave cannot call functions of dynamic libraries provided by the untrusted OS (including system calls to the OS, which are indeed forbidden) [12]. Therefore, all the libraries providing functionalities needed by code running inside the enclave must be statically linked, thus ensuring that their content can be verified during the remote attestation procedure.

For this reason, there exists *library OS* such as *Haven* of *Graphene-SGX* [3, 42] which may execute syscalls on behalf of the enclave, handing in the results. By statically linking these libraries to an enclave, it is possible to port legacy applications to Intel SGX, although reusing code that was not written with the Intel SGX threat model in mind is strongly discouraged. Indeed, side-channel attacks, which surprisingly enough are explicitly excluded by Intel from the threat model of Intel SGX [11], are extremely reliable in extracting secrets if the developer doesn't explicitly address them. Side-channels are unwanted information sources that arise from the physical implementation of a system rather than software or hardware vulnerabilities. The existence of these side channels fostered a rich line of research [2, 3, 42], focusing on legacy applications to showcase the implicit limitations of Intel SGX, but also proposing mitigations [20, 34, 35] to avoid the design of ad-hoc applications.

In the case of Intel SGX, there are two main side channels, namely the sequence of page faults observed by the untrusted OS and the influence of *cache evictions* on memory access time, which allow to infer the *memory access pattern* of an application. This information turns out to be sufficient to exfiltrate some confidential data from the enclave thanks to the little amount of noise exhibited by these side channels, up to the point of defining them *controlled channels* due to their determinism.

Controlled channel attacks primarily exploit the *memory access pattern* of an application to extract secrets during its execution. In fact it is customary that software contains data-dependent memory accesses that can be monitored by a privileged attacker [42]. Algorithm 2.1.1 shows a secret-dependent *control transfer* that happens during the well-known square-and-multiply algorithm to compute the modular exponentiation between an integer $c \in \mathbb{Z}_N$, for a composite N, and the exponent *exp*. This algorithm is employed in vanilla implementations of the decryption procedure of RSA cryptosystem, where the integer c is a ciphertext and the exponent *exp* is the secret key. In this algorithm, the square operation (line 4) is performed in every iteration of the loop, while the multiplication (line 7) is performed only when the current bit of the private exponent is set to 1: whenever an attacker observes two consecutive square operations, she knows that the current bit of the private exponent is a 0, while a multiplication interleaved between two square operations necessarily identifies the current bit of the private exponent as 1. With this trick, the adversary is able to fully reconstruct the private exponent of an Rivest-Shamir-Adleman Cryptoscheme (RSA) key-pair. *Data accesses* are prone to similar attacks. For example, common Advanced Encryption Standard (AES) implementations resort to precalculated T-tables to squash the `SubBytes` and `MixColumns` phases. At the beginning of each round, the 16 bytes composing the AES state are used to index the T-tables. In case of 128-bits keys, the AES state is initialized with the XOR of the plaintext and the key. If the T-tables access pattern (an information which is not protected by Intel SGX) allows to guess the value of the bytes used as index during the first round, a Chosen-Plaintext Attack (CPA) can easily reveal the key by XORing the guessed AES state with the known plaintext [35]: this method can exploit vulnerable binaries and extract up to 25 bits of information in a single round, against the 3 bits of the best known cryptanalytic attack.

Even though these examples refer to leaking cryptographic keys, it is possible to reconstruct different kind of secrets with enough domain knowledge. For example, in [42] the authors attack three widely used libraries: `Freetype`, which renders characters into bitmaps using a font, `Hunspell`, a popular spell-checking library that uses a hash-table based dictionary, and an implementation of Joint Photographic Experts Group (JPEG) transformation, a lossy compression method for digital images, producing a raster representation.

The attacks were delivered using a well-established methodology that leverages the high privileges that the threat model endows. Unlike other

Algorithm 2.1.1: RSA SquareAndMultiply algorithm

```
Input: A message $c \in \mathbb{Z}_N$, for a composite $N$
An RSA private exponent $exp$
Output: $m^{exp}$ mod $N$
1  $res \leftarrow m$
2  $n \leftarrow \lceil \log(exp) \rceil$
3  for $b \leftarrow n - 2$ to 0 do
4      $res \leftarrow res^2$ mod $N$
5      if $exp_b == 1$ // $exp_b$ denotes the $b$-th bit of $exp$
6      then
7          $res \leftarrow res \cdot m$ mod $N$
8      end
9  end
10 return $res$
```

solutions present in literature [12], Intel SGX leaves page tables under the control of the OS, in order to avoid drastic modifications to memory management. As a malicious OS may arbitrarily write on the page table, it can easily trigger page faults on enclave memory accesses in several ways, for instance by resetting the *present* flag for all the virtual pages in `ELRANGE` (see Section 2.1.1). When a *trap* occurs during enclave execution, the logical processor exits enclave mode after saving its execution context to the EPC and cleaning-up the CPU state, a process known as Asynchronous Exit (AEX). The exception is handed to the OS, that performs a potentially malicious page-fault handler. Since the page offset bits are cleared while in enclave mode, an attacker may only monitor memory accesses with the granularity of a page, i.e. 4kB. However, this is sufficient to retrieve secrets if the applications are carefully profiled offline.

By inspecting the code, it is possible to spot where it performs secret-dependent memory accesses either to a function or data. The attacker will record the sequences of page faults that leak secrets. When executing the enclave, she will mark those pages as not present, in order to trigger page faults and compare runtime traces to the sequences she previously collected. However, tracking a subset of enclave pages may yield false positives: in fact, this is equivalent to sample the access patterns only for tracked pages. It may happen that the adversary detects a sequence of page faults that leaks a secret, but in reality the enclave performs other memory accesses to untracked pages, since it is executing a totally different portion of the program instead. Thus, if the attacker is not careful in including in the tracked set all the pages that may be interleaved with a significant sequence of accesses, he may mispredict some secrets.

The authors managed to extract confidential text from the first two libraries and draw the borders of images decoded from JPEG, after wrapping the libraries in enclaves. The attacks were successful in reconstructing up to 70% of the texts and draw the borders of several JPEG pictures up to the point that the portrayed objects are distinguishable.

The drawback of this methodology is that repeatedly triggering page faults incurs in a performance penalty that ranges from $3\times$ to more than $300\times$, possibly raising the suspicion of a careful user.

The discovery of controlled channels has raised the interest in automated techniques to delete information leakage of legacy applications and libraries. In [35] a compiler-based technique known as *deterministic multiplexing* achieves *page-fault obliviousness*, i.e. all the execution paths of a sensitive function trigger the same page fault trace. A sequence of instructions that are executed without branching constitutes a *basic block*: the different execution paths of a function can be arranged in the shape of a tree whose nodes are basic blocks. The execution of a function is equivalent to a tree traversal from the root: when the control flow reaches the end of a basic block, either the left or right child is executed based on the evaluation of a branching condition. While bounded loops are unrolled, unbounded ones are treated as a separate execution tree in order to conceal secret dependent branches that happen within it. For example, in Algorithm 2.1.1, it is necessary to hide whether or not line 7 is executed based on a condition that leaks secrets, but not the overall number of iterations of the loop. The original tree that represents a function might be unbalanced: hence, the first transformation performed by deterministic multiplexing is introducing padding basic blocks to obtain a perfect binary tree. The execution of the function is then performed level-wise: all the basic blocks in the current level are copied to an executable page named *code staging area* to trigger the same sequence of page faults.

The CPU later jumps at the right offset to execute the correct basic block among the ones on the same level of the tree. All the data that might be accessed is copied to a *data staging area*, that is flushed at the end of the execution of a level to take into account possible updates. Deterministic multiplexing is quite expensive and might have a 4000x overhead in execution times. The functions of the program that will undergo the transformation need to be chosen carefully. Even though manual optimizations may reduce the performance penalty, this solution mainly serves as a stopgap [3] that is unlikely to be adopted in contexts where execution times matters.

A different line of research exploits Intel Transactional Synchronization Extensions (TSX). TSX were originally meant to ease the development of multithreaded applications and improve their peformance: instead of acquiring exclusive locks on resources, several transactions are allowed to enter the critical section of a program simultaneously. Transactions commit changes to shared data structures if they don't incur any conflict, performing a rollback otherwise. This makes sure that each transaction has permanent side effects only if it can be executed atomically. To achieve the purpose, the modified data (the so called *write set*) has to fit the private L1 cache of a logical processor, so that local changes are not visible from the other processors. On the other hand, the *read set* needs to fit the L3 cache. TSX strict policy treats cache evictions occurring during a transaction as an error, since it would make temporary changes visible before commitment.

The reason why TSX comes in hand to secure SGX applications is that any exception, including interrupts, aborts a transaction and jumps to a user-space handler rather than notifying the OS. This way, even when a page fault occurs, the application can conceal this event to the OS in the first place, adopting countermeasures against a possible attack. T-SGX [34] uses this behaviour to hide page faults, thus preventing the OS from collecting traces that leak secrets. The core idea is to wrap function calls perfomed in enclave mode in TSX transactions. However, the implementation is not straightforward, since TSX supports only read and write sets that fit, respectively, the L3 and L1 caches and any interrupt will abort transactions that have been running for too long. A possible solution is to divide the code into execution blocks that are likely to complete without benign error conditions. However, since all of these blocks will be wrapped in a separate transaction, page faults may be exposed in the time interval between two consecutive transactions that reside in different memory pages. In order to face this problem, T-SGX employs a single page, called *springboard*, which stores a thunk code which is in charge of starting a transaction and then jumping to the proper code block, that generally resides in a different page. When a code block ends its execution, the address of the next block is placed in a register, such as `%r15`. Control flow is transferred back to the springboard, which ends the current transaction and jumps to the next block. The jump instruction is performed in a new transaction if the next block is labeled as sensitive code whose page faults need to be concealed. It's easy to prove that any page fault directly observable by the OS happens within the springboard: in fact, any attempt to unmap a code page to induce a page fault results in an aborted transaction, while the only page faults that can be triggered outside a transaction are the ones happening within the springboard. When an exception arises, T-SGX employs a simple policy: it restarts the execution of a same block up to 10 consecutive times, before signaling this event as an attack. The number of restarts reflects the common benign abort events that may happen during the execution of a program. The runtime overhead of T-SGX is usually 2x, which seems a good compromise for an automated technique.

T-SGX effectively thwarts contolled channel attacks as presented in [42]. However, the powerful threat model of Intel SGX allows to mount more refined attacks that exploit architectural details of CPUs. Specifically, the Hyper-Threading (HT) technology featured by Intel CPUs allows to concurrently run two threads on the same physical core thanks to partial replication of functional units: each core implementing HT appears to the OS as two separate logic processors, that can be scheduled independently. However, some of the resources of the core are not duplicated, notably caches. The core idea of *cache attacks* on Intel SGX is mapping on the same physical core a thread executing in enclave mode and a thread controlled by the adversary, referred to as *spying thread*, and then to exploit cache contention between these threads in order to extract secret information. The privileged attacker cannot trivially exfiltrate information from contended caches: indeed, since caches are totally transparent to software, there is no way to directly access them. Furthermore, a spying thread cannot access memory frames in the EPC since the TLBs are split: even if the enclave thread caches a virtual to physical mapping referring to the EPC, the spying thread needs to pass all the security checks implemented by the PMH by itself. However, when two threads reside on the same physical core they contend the L1 cache, and it may happen that the execution of a thread causes the eviction of cache lines owned by the concurrent thread. This will produce the evident side effect of a slower memory access, since the required data will be fetched from lower levels of the memory hierarchy. By measuring memory access times, the spying thread can infer useful side-channel information.

The attack described in [3] exploits a novel technique known as FLUSH + FLUSH. An `x86` instruction, `clflush`, forces the eviction of all the cache lines that correspond to a specific virtual address. Its pecularity is that it takes more time to execute when entries corresponding to that virtual address are found in the cache, while it's faster otherwise. The attack manages to collect information about the memory access pattern by monitoring page tables. In fact, when the PMH performs address translation, it caches the chunks of page table it uses. A root level adversary may `clflush` the address of the portion of the page table that refers to the code or data it wants to monitor during the execution of the enclave. When `clflush` takes a longer time to execute, this means that the PMH performed a page table walk on behalf of the enclave, and that the enclave accessed a releavant page that leaks secrets. Since cache lines are 64 B wide, they contain 8 page table entries and thus allow to monitor the access pattern at the granularity of 32 kB. An alternative approach consists in monitoring the *accessed* and *dirty* bits (`A/D bits`) of page table entries, that are set whenever a read or write operation occur. The TLB is always flushed on enclave exit, hence an enclave always triggers page table walks when it starts its execution. With these methods, an attacker can monitor the memory access pattern even when instrumenting code with T-SGX, as page tables are managed by the OS and not included in the read/write set of a transaction.

The spying thread monitors the access to specific *trigger* pages, and sends an interrupt to the enclave when an access is detected. It then inspects the *accessed* bits of the page table in order to determine which pages the enclave required within two interrupts. Since T-SGX allows a transaction to restart up to 10 times, a single interrupt is not detected as an attack Even though the information the attacker can collect is not as complete as a full page fault trace, in most cases it is sufficient to distinguish secret dependent branches. The authors validated the attacks leaking a 512-bit private key from the double-and-add routine of libgcrypt, used for elliptic curve cryptography. This function exhibits the same control flow as Algorithm 2.1.1, save for the fact that square operations and multiplications are substituted by doubling operations and additions, respectively. The `A/D bit` technique allows to retrieve a full 512-bit private key while the Flush+Flush approach up to 485 bits, enough to complete the attack by means of bruteforce. Flush+Flush technique can monitor accesses to virtual pages with a finer instruction granularity because the `clflush` instruction can be executed in a much tighter loop. However, it allows to detect access to trigger pages at the much coarser granularity of 32 kB, while the slower `A/D bits` technique achieves a resolution of 4 kB.

Both the approaches apply when protecting the enclaves via T-SGX. Indeed, they don't require page fault traces and frequent interrupts.

A different cache attack, named Prime+Probe [2], allows to totally cut necessity for interrupts. In this setting, the spying thread pollutes all the L1 cache lines with dummy data and then executes the enclave. By carefully measuring access time to the dummy data, it understands which entries were evicted due to the execution of the enclaved thread. Since the information that can be collected primarily amounts to the cache sets that were partially evicted by the enclave, only specific applications with regular access patterns to arrays are vulnerable. Notably, implementations of cryptographic primitives, such as RSA that rely on precalculated tables and T-table based implementation of AES are prone to this attack. Prime+Probe suffers from relevant noise and requires several runs to obtain reliable results. This condition is not easily achieved in many contexts because of remote attestation, that allows to establish a secure channel between the enclave and the user. Since the ephemeral symmetric key is safely stored inside the enclave and destroyed as soon as the session is torn down, performing a replay attack to collect enough statistics about the execution of the enclave is not always possible.

In order to defeat cache attacks, Cloak [20] tries to preload sensitive portions of the program in the cache after starting a TSX transaction. Since read or write set evictions result in a roll back, a transaction can only succeed if its memory accesses hit the cache during its whole execution. Even if TSX does not encompass the L1 instruction cache in the read set of a transaction, the authors empirically verified that instruction cache evictions caused by a concurrent thread trigger transaction failure. This undocumented behaviour conceptually allows to preload the L1 instruction cache as well. In order to achieve this in practice, they devised a low level trick. `x86_64` supports multibyte `NOP` instructions, for example, the opcode `0x0f1f8000000000`. The `NOP` is recognized only by its prefix `0x0f1f80`, while the remaining zeroes are padding and can be overwritten. Suppose that the address of the first zero byte of the `NOP` is stored in `%r14`. Let the current instruction be `jmp %r14`; `%r15` stores the address of the next instruction, instead. We encode `jmp %r15 = 0x41ffe6` in three of the 4 spare bytes of the `NOP`, obtaining `0x0f1f8041ffe600`. When we execute that assembly, the CPU first jumps to the location pointed by `%r14`. The `NOP` and the following 64B of code will be loaded in the instruction cache. Since we don't execute the `NOP`, but rather `jmp %r15`, we jump back to the previous location. When we happen to execute the code in that cache line, the CPU will stumble in a `NOP` and ignore padding bytes, thus skipping the instruction encoded in the `NOP`. This approach doesn't use the stack, in order to avoid evictions of the data cache.

Cloak defeats PRIME+PROBE attacks. In fact, the spying thread that tries to read its dummy data to measure the access time triggers evictions of data in the write set of the running transaction. Since PRIME+PROBE is already very noisy and unreliable, success rates drop to zero when code is instrumented with Cloak. Cloak was used to shield vulnerable implementations of cryptographic primitives, namely the already cited RSA exponentiation and T-tables based AES encryption, as well as a tree-based classification algorithm, exhibiting an overhead that didn't exceed 30x. Notably, it sometimes improved performance, due to the fact that it forces all the sensitive computation to happen in the cache, thus improving memory access times.

On the other hand, Cloak is totally inneffective against the FLUSH+FLUSH attack we described. In fact, in that case the cache side channel is used to infer which page table entries were brought in the L1 as a consequence of a page table walk. Since page tables are transparently managed by the OS and PMH, they don't fall in the read/write set of any transaction running within the enclave.

Pure cache side channels, like the PRIME+PROBE technique, and memory access pattern leakage are mostly orthogonal. In fact, the latter aim at discovering secret dependent branches of an application. On the other hand, cache attacks provide a fine grained analysis of the access patterns of a program, but their noise and inaccuracy requires several runs of an application to collect enough statistics. T-SGX [34] effectively thwarts basic controlled channel attacks, but is defeated by attacks that exploit cache side channels to extract information about page table walks. On the other hand, while deterministic multiplexing [35] provides page fault obliviousness, the granularity of probabilistic cache attacks such as PRIME+PROBE [2] may leak memory accesses in the *data staging area.* Cloak [20], instead, defeats cache side channels, but since page table walks are not wrapped within a TSX transaction, it is vulnerable to the `A/D bit` techniques as well as FLUSH+FLUSH [3].

Combining several countermeasures may incur in a great performance overhead and some technologies such as TSX may not be enabled or supported on all platforms, thus disabling many of the presented shielding techniques. Even if the scope of this work is limited to memory attacks, they are not the only concern of Intel SGX. Foreshadow [39] exploits cache side channels and out-of-order execution quirks to leak sensitive secrets, notably the private keys used by the Quoting Enclave (QE) to create attestation reports. This attack totally breaks the remote attestation mechanism, thus dismantling the security guarantees of Intel SGX.

The variety and effectiveness of the attacks we have just presented highlight the need to put aside the idea of retrofitting legacy applications to Intel SGX, and Intel's guideline to align all sensitive information to a 4 kB seems a simplistic statement given the presence of attacks that can leak access patterns at a much finer granularity. In this work, we focus on an alternative approach, which tries to prevent attacks by re-designing applications to remove any dependency between memory accesses and the sensitive data. In particular, the control flow and data access pattern of an *oblivious* algorithm does not depend at all on input data. Page table traces are uniform during every execution, while cache attacks are defeated by breaking the correspondence between the secret indices used to access data structures and the resulting memory addresses. All the presented countermeasures are hence unnecessary because the application would exhibit a totally flat and uniform execution flow by itself, and hence an attacker cannot gain any knowledge during its execution.

2.2 Oblivious RAM

Generally speaking, memory is a contiguous array of homogenous entries that are indexed via an ordinal number, that tells the absolute position of each element. Besides providing the necessary information to access a location, the index also represents the logical identifier of each memory cell. A Random Access Memory (RAM) allows to directly fetch the i-th element without sequentially scanning the preceding $i-1$ ones, which is required by sequential memories instead.

The correspondence between logical and physical identifiers eases the implementation and usage of memories. Even when adopting address translation and other advanced memory management techniques, these basic principles still hold: algorithms access data structures using logical indices, such as integers, that are trivially mapped to virtual addresses. Nevertheless, this simple mapping allows to infer the sequence of logical identifiers accessed by the application, referred to as *logical access pattern*, from the sequence of physical identifiers accessed by the application, referred to as *memory access pattern*, which turns out to be a relevant security issue when the memory content must be hidden from an adversary who can observe the latter. Indeed, the memory access pattern can be analyzed to extract meaningful information about the logical access pattern of an application, which in turn may depend on sensitive information. This security issue arises in several application scenarios, the most common one being outsourced data storage. In this context, there are two entities: a client, with limited storage capabilities, who owns a a big dataset which contains private information, and a remote untrusted server, which has much significant storage capabilities. Given its limited storage capabilities, the client offloads it to the server, which exposes an Access procedure to fetch and update the content of the outsourced data. In order to preserve the confidentiality of the data, the client encrypts the dataset before offloading it; nevertheless, by observing the memory access pattern of the client, the remote server will be able to infer the logical access pattern of the client, which may leak sensitive information about the outsourced data. This leakage can be avoided by breaking the correlation between logical indices and memory access patterns: a trivial solution would be always accessing all the elements to hide which one is actually needed, but it scales poorly with the size of the dataset. This is where Oblivious RAMs (ORAMs) come into place. ORAMs are cryptographic primitives which decouple the logical identifiers of memory locations from their addresses, resorting to continuous scrambling and re- encryptions to break any correlation that may arise in subsequent accesses. In ORAM protocols, the dataset is stored by the remote server in an ORAM data structure; the server does not need to provide a significant computational power: its task simply amounts to fetching and writing back the required data, while the scrambling and encryption operations happen on the client side. Data fetched by the client is exchanged through the network, so the bandwidth blowup is the major figure of merit in evaluating remote ORAM protocols. While the first ORAM schemes were impractical due to their complexity, the Path ORAM framework presented in [36] has paved the way for their adoption in several applications that provide strong security guarantees while incurring reasonable runtime penalty. We now describe Path ORAM as well as two improved constructions, namely Ring ORAM and Circuit ORAM, which will be employed in this work.

2.2.1 Path ORAM

Path ORAM [36] received a lot of attention due to its simplicity, security and performance.

The basic storage unit is called *block* or *record*: its size B varies according to the data it will contain. Therefore, a dataset with l bits is split in $N = \lceil \frac{l}{B} \rceil$ ORAM blocks. Several blocks are grouped together to make up a *bucket*. The parameter Z defines the number of records contained in a bucket.

The Path ORAM is a complete binary tree of buckets with $L = \lceil log_2 N \rceil$ levels numbered from 0 (root) to $L - 1$ (leaves). Figure 2.3 represents the ORAM tree for $N \in [5, 8]$ and $Z = 4$. As the ORAM contains $2^L \geq N$ buckets, each containing Z blocks, it is straightforward to observe that there are more than N blocks in the whole ORAM: thus, there are some blocks, referred to as *dummy*, which store random data instead of a portion of the original data. These additional blocks are necessary to aid the scrambling operations intended to decouple the position of a block in the ORAM tree

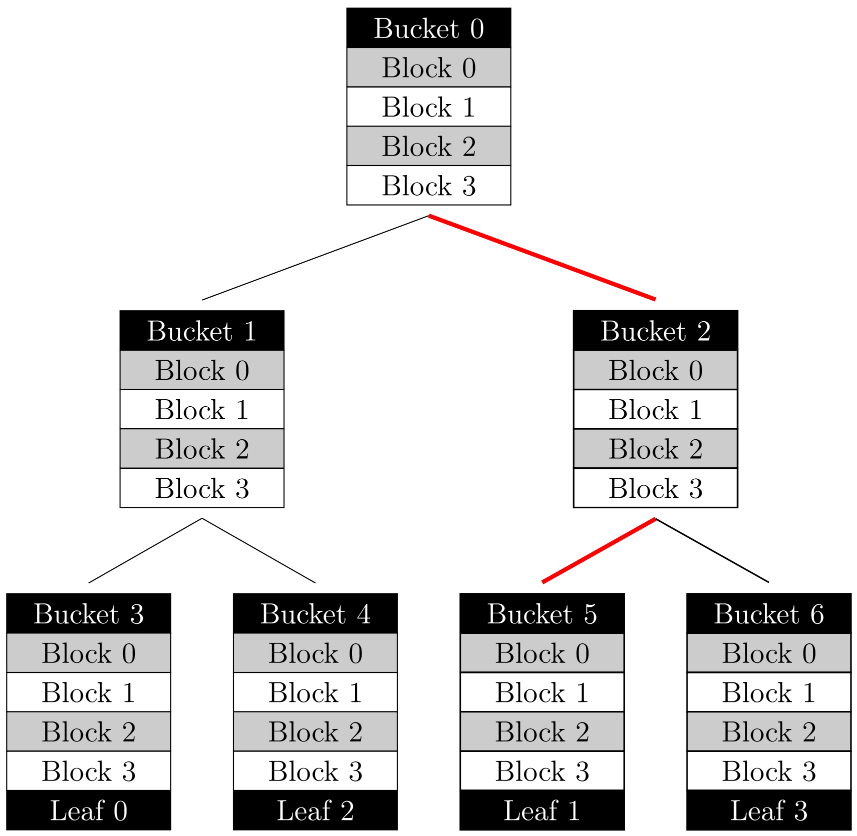

Figure 2.3: A Path ORAM with $Z = 4$. The path leading to leaf 1 is colored in red

from its logical identifier, which is called *block id*, or *bid* in short. A dummy block is tagged with an empty id, that is $bid = \perp$.

The leaves are numbered in *reverse lexicographical order* by tagging each left branch along its path from the root with 0 and each right branch with 1. For example to follow the red path, it is necessary to traverse a right and then a left edge, generating the sequence 10. Once reversed, it becomes 01, which corresponds to the *leaf id*, shortly denoted as *lid*, shown in the picture. Thus, each $lid \in [0, 2^{L-1} - 1]$ identifies a single path from the root to the leaf. Performing the tree exploration from the LSB of the leaf id will come in hand when describing the more elaborate Ring and Circuit ORAM protocols. In order to adopt a uniform convention for all the ORAMs, this leaf labelling method can be used for Path ORAM as well.

Each block, apart for its block id and its data, also contains the leaf id of the path where it resides. Leaf ids play a fundamental role in the retrieval of data blocks, since the following invariant holds in Path ORAM: each data block resides in one of the buckets that belong to the path identified by its leaf id. As a consequence, when a block is stored in a non-leaf bucket, its leaf id corresponds to one of the leaves of the subtree rooted at that bucket. The client keeps track of the mapping between a block id and its corresponding leaf id in the *position map*, denoted as pos_map in the algorithms. In addition, the client locally holds a list of blocks which cannot be currently inserted in the ORAM tree without breaking the invariant property, which is called *stash*. We now describe the ACCESS procedure of Path ORAM to retrieve a block with identifier *bid* from the ORAM tree. The client retrieves the leaf id from the position map: $lid = pos\_map[bid]$. It then asks the server to return the whole path corresponding to *lid*, appending all the blocks in the path to the stash. The whole path on the ORAM tree is invalidated and will be updated at the end of the procedure. Then, the client looks for a block in the fetched path with tag *bid*. After that, it samples a random value lid' and updates with these value both $pos\_map[bid]$ and the leaf id in the record metadata. In this way, subsequent accesses to the same block will fetch a different path on the ORAM tree. The new path has no correlation with the older one, given that the mapping between the block ids and the leaf ids is not leaked to the server. To keep the stash size small, the client tries to move as many elements as possible from the stash to the ORAM tree, a procedure which is called *stash eviction*, or simply *eviction*. In order to avoid that the leaf ids of the blocks in the stash are leaked, the blocks are evicted from the stash only if they can be placed in the path which has just been fetched by the client. Therefore, in order to preserve the invariant that a block is kept in the path that leads to its leaf id, the client tries to evict the updated record in an empty block contained in a bucket that is a common ancestor of *lid* and lid'. The stash eviction is performed from the leaf to the root in order to push as down as possible the elements contained in the stash. In fact buckets placed at higher levels of the tree are common ancestors of an increasing number of leaves, and thus they can host many more blocks and improve the eviction probability. After placing as many blocks in the path as possible, the newly constructed path is re-encrypted in order to hide which data was modified, and is written back to the ORAM, in place of the previously invalidated path. A fairly detailed description of the operations performed by the client during an ORAM access is described in Algorithms 2.2.1 and 2.2.2, split between the block fetch and path eviction. FetchBucket and WriteBucket are remotely executed on the server and respectively read and write the bucket at a certain level of a specified path. GetMaxDepth evaluates the maximum depth where a block can reside given its leaf id and the current eviction path: if the value it returns is greater or equal than the current depth, then the block can reside in the corresponding bucket without breaking the leaf id invariant.

Algorithm 2.2.1: Path ORAM Fetch

```
Input: A block id $bid$
Input: $new\_data$ to write to the block, it may be $\perp$ if it's a read
       operation
Output: The $data$ corresponding to block $bid$ and its leaf id $lid$
$data \leftarrow \perp$
$lid \leftarrow pos\_map[bid]$
$new\_lid \leftarrow$ UniformRandom$(0, 2^{L-1} - 1)$
$pos\_map[bid] \leftarrow new\_lid$
// look for entry bid in the path lid
for $l \leftarrow 0$ to $L - 1$ do
    // fetch the bucket at level l from path lid
    // bucket is an array of blocks
    $bucket \leftarrow$ FetchBucket$(lid, l)$
    $stash \leftarrow stash \cup bucket$
end
foreach $block \in stash$ do
    if $block.bid == bid$ then
        $data \leftarrow block.payload$
        $block.lid \leftarrow new\_lid$
        break
    end
end
return $(data, lid)$
```

As the client has limited storage capabilities, it is necessary to guarantee that the stash size does not exceed its memory limits. To this extent, an upper bound on the stash size S is set: if more than S blocks must be stored in the stash, then a stash *overflow* event occurs, which is a failure condition of

Algorithm 2.2.2: Path ORAM EVICT

```
Input: The $lid$ of the path to evict
for $l \leftarrow L-1$ to $0$ do
    $bucket \leftarrow \perp$
    $i \leftarrow 0$
    foreach $block \in stash$ do
        /* if bucket at level l is a common ancestor of
           block.lid and lid                                    */
        if GETMAXDEPTH($lid, block.lid$) $\geq l$ then
            $bucket[i] \leftarrow block$
            $stash = stash - block$
            $i \leftarrow i + 1$
        end
        if $i == Z$ then
            break
        end
    end
    WRITEBUCKET($bucket, lid, l$)
end
```

Algorithm 2.2.3: Path ORAM ACCESS

```
Input: A block id $bid$
Input: $new\_data$ to write to the block, it may be $\perp$ if it's a read
       operation
Output: The $data$ corresponding to block $bid$
($data, lid$) $\leftarrow$ FETCH($bid, new\_data$)
EVICT($lid$)
return $data$
```

the ORAM. Even worse, in case of a stash overflow, the server comes to know that $S+1$ valid blocks are in the stash, thus causing a slight information leakage. Path ORAM provides upper bounds on the overflow probability for each combination of Z and S for $Z > 5$. The overflow probability of a Path ORAM with stash size S is $\Theta(1.6^{-S})$ [36]. Empirically, $Z = 4$ is the minimum value for which the theoretic bounds still hold and the stash size is independent of the number of elements N stored in the ORAM.

The bandwidth overhead of an ORAM is defined as the ratio of exchanged data and block size B. Every time a path is fetched, $B \cdot Z \cdot \log N$ bits are exchanged between the client and the server, thus leading to a bandwidth of $\mathcal{O}(Z \cdot \log N)$.

2.2.2 Ring ORAM

Ring ORAM [16,28] introduces critical modifications to the eviction procedure of the Path ORAM to improve its bandwidth. While the overall structure and mechanism of the ORAM is left unchanged, the bucket structure is heavily modified to support the optimizations it introduces. Moreover, the fully optimized version requires a server capable of performing simple computations, even though this is not strictly necessary.

The bandwidth overhead of Path ORAM is $\mathcal{O}(Z \cdot \log N)$ while Ring ORAM manages to reduce the online bandwidth to fetch a block to $\mathcal{O}(1)$. The first observation to achieve the bandwidth reduction is that evictions cost $\mathcal{O}(Z \cdot \log N)$ by themselves, and thus cannot be performed for every access. The *eviction period* A rules how many accesses on the ORAM are performed before full eviction takes place. In Path ORAM, an eviction is performed after every access, thus $A = 1$. Conversely, Ring ORAM achieves higher eviction periods by employing a fixed eviction schedule to uniformly distribute blocks along paths. Specifically, the paths to be evicted are chosen by iterating over the leaves of the tree with increasing values of the leaf id; as each leaf corresponds to a path in the ORAM tree, this schedule allows to eventually evict all paths in the tree. The advantage of this deterministic eviction schedule ensures is that every time an evicted path includes a bucket, it will include its left or right child in an alternated fashion, to maximize the amount of blocks that are flushed from their parent bucket.

While the overall bandwidth of Ring ORAM remains the same, the cost of evictions is amortised over many rounds, yielding visible practical improvements: Path ORAM generates 2.6× more traffic than Ring ORAM [28]. This approach allows to achieve a much better online bandwidth, when coupled with some tricks in order to drop the bandwidth required to fetch an element from $\mathcal{O}(Z \cdot \log(N))$ to $\mathcal{O}(1)$.

To achieve this bandwidth reduction, each bucket is enriched with additional metadata, which are reported in Table 2.1. The size of these metadata is negligible with respect to the block size B, therefore downloading all the metadata of a path requires significantly less bandwidth than downloading the path itself. We now show how these metadata can be employed by the client in order to reduce the bandwidth to fetch the requested block.

The client first downloads the metadata of all the buckets in a path, which contain information about the blocks that a bucket contains. Unlike Path ORAM, which downloads all the Z records in a bucker, the client of Ring ORAM picks only one block from each bucket: if the requested block is found, the client selects it, otherwise it chooses one at random. The records that have already been accessed are marked as invalid, in order to prevent correlation attacks across ORAM accesses. However, in this way $\log N$ potentially valid blocks would be added to the stash. Since evictions are not performed every round, this would cause the stash size to grow fast, overflowing the limited storage capabilities of the client. This pitfall can be overcome if all the blocks, save for the one that the client wants to access, are dummies. Hence each bucket also contains D records that are guaranteed to be dummy. The Z possibly non-dummy blocks are permuted to prevent the server from distinguishing dummy records from the valid ones. The bucket metadata stores the Pseudo-Random Permutation (PRP) in encrypted form, along with the block ids of the Z valid blocks. To maintain the invariant that at most one valid block is fetched from a path, every bucket is reshuffled after D accesses to refresh the dummy blocks: this operation, that is named EarlyReshuffle, guarantees that a bucket never runs out of dummy records, and is invoked each time the counter of accesses to a bucket hits D. We highlight that the validity bits of the records contained in a bucket, along with the total number of accesses before an EarlyReshuffle, are stored plaintext metadata even in unprotected memory. In fact, not only the server knows how many times a bucket is accessed and which record is

Table 2.1: Format of the bucket metadata

Field	**Bit width**	**Elements**	**Description**
IV	λ	1	IV of given security parameter
Block id	$\log N$	Z	Block ids of the Z valid blocks
Leaf id	$\log (2^{L-1} - 1)$	Z	Leaf ids of of the Z valid blocks
PRP	$\log (Z + D)$	Z	Index of the valid blocks in the bucket
Valid	1	$Z + D$	Set if the corresponding record has been fetched
Counter	$\log D$	1	Counts the number of accesses to the block

selected from time to time, but also cooperates in updating these portions of bucket metadata. Therefore, the server is a *honest but curious* actor that does not tamper with these values, but collaborates with the client to implement the protocol.

In order to allow the decryption of each record independently, Ring ORAM adopts a block cipher in Counter Mode (CTR Mode). The Initialization Vector (IV) is contained in the bucket metadata, as it is a public parameter. The offset of a block in the bucket serves as a way to derive the right counter to encrypt the block itself. This first trick allows to erase the Z factor from the bandwidth, as the client requests only one block from each bucket in a path.

The so called *XOR Technique* further reduces the bandwidth by a $\log (N)$ factor, thus obtaining $\mathcal{O}(1)$ online bandwidth. Suppose that the dummy blocks are all ciphertexts encrypting a plaintext value d fixed by the client and unknown to the server. In this case, the client can reconstruct locally the dummy record d_i as it appears in the i-th bucket: indeed, d_i can be computed by encrypting d in CTR Mode mode with a counter CTR_i which is derived from the IV of the i-th bucket and the offset of the dummy block in the bucket. Suppose that the x-th bucket is the one containing the target record. Therefore, the server can return a single block obtained by XORing together all the blocks in a path, which is equivalent to a random masking of the target block:

$$b = \bigoplus_{i=0}^{L-1} b_i = \left(\bigoplus_{i=0, i \neq x}^{L-1} b_i \right) \oplus b_x = mask \oplus b_x \tag{2.1}$$

where b_x is the real block among the L blocks chosen by the client. As the client can reconstruct the dummy records, it can also reconstruct the *mask*, thus retrieving the block b_x:

$$\left(\bigoplus_{i=0, i \neq x}^{L-1} d_i \right) \oplus b = mask \oplus b = b_x \tag{2.2}$$

This way, the correct block can be retrieved by sending only B bits over the network, thus reaching $\mathcal{O}(1)$ online bandwidth.

The ACCESS procedure of ring ORAM is extensively showed in Algorithm 2.2.4. While its overall structure is the same as Path ORAM, we need some refinements to account for the modified eviction procedure and the bucket *reshuffle*. In the first phase, the client fetches the metadata of all buckets in a path by the FETCHBUCKETMETADATA function. Then, it employs the SELECTOFFSET (detailed in Algorithm 2.2.5) to choose the block to be fetched from each of the L buckets in the path. The server employs these offsets to retrieve the blocks and computes a single block by the XOR trick, which is sent to the client. The eviction procedure reads the whole path identified by *eviction_path* and performs the same operations as the Path ORAM, save for the fact that, after eviction, the client generates a PRP for each bucket and shuffles the bucket accordingly. Lastly, the EARLYRESHUFFLE scans the metadata of the fetched path and determines the buckets which needs to be re-shuffled, i.e., all the buckets whose *bucket.counter* is greater than D. the client fetches all these buckets from the ORAM tree, it adds all their valid and non-dummy entries to the stash and it refills the buckets, uploading them once permuted.

The Ring ORAM features many parameters, summarised in Table 2.2. The formal proof of their relationship is treated in [16], here we just present how to derive them. Once again the stash overflow probability decreases exponentially with stash size.

Ring ORAM is an elaborate construction, and even if the basic working

Algorithm 2.2.4: Ring ORAM ACCESS function as implemented on the client

```
Input: A block id $bid$
Input: $new\_data$ to write to the block, it may be $\perp$ if it's a read
       operation
Output: The $data$ corresponding to block $bid$
persistent variable $access\_counter \leftarrow 0$
persistent variable $eviction\_path \leftarrow 0$
$lid \leftarrow pos\_map[bid]$
$new\_lid \leftarrow$ UniformRandom$(0, 2^{L-1} - 1)$
$pos\_map[bid] \leftarrow new\_lid$
for $l \leftarrow 0$ to $L - 1$ do
    // meta is a struct as described in Table 2.1
    $meta \leftarrow$ FetchBucketMetadata$(lid, l)$
    $(offset[l], dummy[l]) \leftarrow$ SelectOffset$(meta, bid)$
end
/* fetch the blocks from the server and perform the pseudo
   XOR trick                                                */
$data \leftarrow$ XorTrick$(lid, offset, dummy)$
if $data == \perp$ then
    $data \leftarrow$ StashFind$(bid)$
    StashRemove$(bid)$
end
if $new\_data \neq \perp$ then
    $stash \leftarrow stash \cup (bid, new\_lid, new\_data)$
end
else
    $stash \leftarrow stash \cup (bid, new\_lid, data)$
end
// perform stash eviction
if $access\_counter$ mod $A = 0$ then
    EvictPath$(eviction\_path)$
    $eviction\_path \leftarrow eviction\_path + 1$
end
$access\_counter \leftarrow access\_counter + 1$
EarlyReshuffle$(lid)$
return $data$
```

Algorithm 2.2.5: Ring ORAM SelectOffset function

```
Input: A block id $bid$
Input: Encrypted metadata $meta$ of a bucket
Output: The chosen offset in the bucket and whether it is a dummy

1  $dec\_meta \leftarrow Dec(meta)$
2  $dec\_meta.counter \leftarrow dec\_meta.counter + 1$
   /* find the indices of dummies by discarding the ones in
      the PRP                                                    */
3  $dummies \leftarrow [0, Z + D - 1] - dec\_meta.prp$
   // dicard dummies that were already fetched
4  $dummies \leftarrow \{index \in dummies \mid dec\_meta.valid[index] == 1\}$
   // select a random dummy
5  $offset \leftarrow dummies[\text{UniformRandom}(0, |dummies| - 1)]$
6  $is\_dummy \leftarrow$ true
7  for $i \leftarrow 0$ to $Z$ do
8      if $dec\_meta.bid[i] == bid \land dec\_meta.valid[dec\_meta.prp[i]]$
       then
           /* if the bid is matched but it is invalid, the
              required data was already fetched and can be found
              in the stash                                       */
9          $offset \leftarrow dec\_meta.prp[i]$
10         $is\_dummy \leftarrow$ false
11     end
12 end
13 return $(offset, is\_dummy)$
```

Table 2.2: Parametrization of the Ring ORAM

Parameter	Description	Value
Z	Valid blocks	Free parameter, ≥ 4
A	Eviction rate	$\arg\max_A Z \cdot \ln\left(\frac{2Z}{A}\right) + \frac{A}{2} - Z - \ln(4)$[1]
D	Dummy blocks	minimize $\left(\frac{2Z}{S} \cdot (1 + Q(A+1, S))\right)$
S	Stash size	$P[\text{overflow}] < \frac{A^S}{(2Z)^S(1-e^{-q})}$[2]

[1] $Q(a, x)$ is the regularized *incomplete gamma function*

[2] $q = Z \cdot \ln\left(\frac{2Z}{A}\right) + \frac{A}{2} - Z - \ln(4)$

principle is the same as Path ORAM, it introduces many tweaks to improve its bandwidth requirements, that is the major source of latency in the remote scenario.

2.2.3 Circuit ORAM

Circuit ORAM is another variation of Path ORAM that introduces a totally different eviction procedure. Circuit ORAM is mainly optimized for hardware implementations [40], focusing on reducing as much as possible the stash size S, the number Z of non-dummy blocks in each bucket and the complexity of the circuit required for the eviction.

The main purpose of Circuit ORAM is to minimize the number of scans of the stash and the fetched path during the eviction. The algorithm frees the stash entry holding the block that can reside in the deepest bucket of the path. This record is carried along the eviction path, from root to leaf, until either the algorithm finds a new block that can go deeper, or it reaches the maximum depth for the current block: in the first case, the two blocks are swapped, while in the second, the current record replaces a dummy one. This process goes on until it reaches the leaf.

Nonetheless, there is a great pitfall in this strategy: while carrying down a block towards its target bucket, it is possible that there are no free entries in the target bucket to host it, which means that it needs to be written back to the stash. To avoid this issue, the algorithm needs to determine the destination of the currently evicted block by considering both the deepest bucket which can host it and and the fullness of the buckets. In this way, it is possible to decide beforehand if it is better to leave the block in a less deep bucket. To this extent, the eviction procedure does two preliminary sweeps on the evicted path metadata to determine which blocks should be moved as deep as possible, to avoid that a block must be written back to the bucket it belong or the stash, which would make an eviction pass totally worthless. Then, it does a last sweep where the blocks are effectively moved in the target buckets.

The first sweep is done by the PrepareDeepest function, reported in Algorithm 2.2.6. PrepareDeepest starts by finding the block of the stash that can reside in the deepest bucket. The stash is considered a bucket placed at level -1, right before the root of the ORAM tree. Then, the buckets are

Algorithm 2.2.6: Circuit ORAM PrepareDeepest

```
Input: A list of buckets $bucket$
Input: The leaf id $lid$ of the current path
Output: An array $deepest$

// the stash is considered a bucket at level -1
$src \leftarrow \perp$
$deepest\_level \leftarrow -2$
foreach $block \in stash$ do
    /* GetMaxDepth evaluated the maximum depth in the
       current path where a block can be moved                */
    $d \leftarrow$ GetMaxDepth($block.lid, lid$)
    if $d > deepest\_level$ then
        $src \leftarrow -1$
        $deepest\_level \leftarrow d$
    end
end
$deepest[-1] \leftarrow \perp$
for $l \leftarrow 0$ to $L-1$ do
    /* record the level of the deepest block that can reside
       in the current bucket                                  */
    $deepest[l] \leftarrow src$
    for $i \leftarrow 0$ to $Z-1$ do
        $d \leftarrow$ GetMaxDepth($bucket[l].block[i].lid, lid$)
        if $bucket[l].block[i].bid \neq \perp \wedge d > deepest\_level$ then
            $deepest\_level \leftarrow d$
            $src \leftarrow l$
        end
    end
    if $deepest\_level < l$ then
        $src \leftarrow \perp$
    end
end
return $deepest$
```

scanned one by one from the root to the leaf. When the i-th level of the tree is considered, the variable *src* holds which of the previous $i-1$ buckets contains the block that can travel deeper in the path, while *deepest_level* is equal to the max depth it can reach. Hence, *deepest*[i] will be assigned the source level of the deepest block found so far, or $\perp$ if no viable blocks exist. Line 15 updates the variable *src* only if a bucket contains a block whose depth is $\geq$ *deepest_level*. Since the buckets are scanned from root to leaf, this policy favours the levels that are closer to the root, which are more valuable since they can contain blocks assigned to a greater number of leaves. The helper function GetMaxDepth finds the level of the deepest common ancestor of two leaves and can be implemented in $\mathcal{O}(1)$ if specific machine instructions are available, or $\mathcal{O}(\log N)$ otherwise.

The second sweep of the eviction is done by the PrepareTarget function, which is reported in Algorithm 2.2.7. For each bucket i starting from the leaf, PrepareTarget reads the correspoding *deepest*[i] entry to test whether or not there is a block in the upper buckets that can reside in the current one. That block will be evicted to the current bucket either if it already has a free block, or if it contains a block that will be moved down, thus making room for a new record. When one of this condition holds, *dst* is assigned the current bucket level, while *src* the value of *deepest*[i]. Once that *src* and *dst* are set, all the buckets between *dst* and *src* are skipped. In fact, the algorithm has already managed to free a block of the more valuable bucket at level *src*. When getting to *src*, *target*[*src*] will be set to *dst* (line 5), meaning that the deepest block in that bucket will be picked up and moved to the level *dst*.

Finally, the third sweep scans the path from root to leaf and evicts the deepest blocks of each bucket to the level pointed by `target`. Eventually, the whole path is re-encrypted and written back to the ORAM tree. The whole eviction procedure is reported in Algorithm 2.2.8.

This eviction procedure simply tries to move the most promising blocks as deep as possible, freeing slots that are closer to the root. However, at most one entry at a time is evicted from the stash, whose size is doomed to increase monotonically as each access appends exactly one block to it. To face this problem, the eviction procedure is applied to two different paths, according to the same deterministic schedule used by Ring ORAM. The usage of such

Algorithm 2.2.7: Circuit ORAM PREPARETARGET

```
Input: A list of buckets $bucket$
Input: The leaf id $lid$ of the current path
Input: The output of Algorithm 2.2.6 $deepest$
Output: An array $target$
// source level of the block to evict
$src \leftarrow \perp$
// destination level of the block to evict
$dst \leftarrow \perp$
for $l \leftarrow L-1$ to $0$ do
    /* if destination was reached, write the src to the
       target array                                          */
    if $l == src$ then
        $target[l] \leftarrow dst$
        $dst \leftarrow \perp$
        $src \leftarrow \perp$
    end
    else
        $target[l] \leftarrow \perp$
    end
    /* bucket.HasFreeBlock returns true if the bucket
       contains a dummy record                               */
    $has\_free\_block \leftarrow bucket[l].\textsc{HasFreeBlock}()$
    /* you can evict a block to the current bucket if it has
       a free block or it will evict a block downwards:
       target[l] $\neq \perp$                                */
    if $(dst = \perp \wedge has\_free\_block \vee target[l] \neq \perp) \wedge deepest[l] \neq \perp$
    then
        $dst \leftarrow l$
        $src \leftarrow deepest[l]$
    end
end
// manage the stash
$target[-1] \leftarrow dst$
return $target$
```

Algorithm 2.2.8: Circuit ORAM Eviction

```
Input: A list of buckets $bucket$
Input: The leaf id $lid$ of the current path
$deepest \leftarrow$ PrepareDeepest($bucket, lid$)
$target \leftarrow$ PrepareTarget($bucket, lid, deepest$)
$hold \leftarrow \perp$
$dst \leftarrow \perp$
if $S \neq \varnothing \wedge target[-1] \neq \perp$ then
    $hold \leftarrow \arg\max_{s \in stash}$ GetMaxDepth($lid, s.lid$)
    $stash \leftarrow stash - hold$
    $dst \leftarrow target[-1]$
end
for $l \leftarrow 0$ to $L - 1$ do
    $to\_write \leftarrow \perp$
    if $l = dst \wedge hold \neq \perp$ then
        $to\_write \leftarrow hold$
        $hold \leftarrow \perp$
        $dst \leftarrow \perp$
    end
    if $target[l] \neq \perp$ then
        $index \leftarrow$
        $\arg\max_{i \in [0,Z)}$ GetMaxDepth($lid, bucket[l].block[i].lid$)
        $hold \leftarrow bucket[l].block[index]$
        $bucket[l].block[index].bid \leftarrow \perp$
        $dst \leftarrow target[l]$
    end
    if $to\_write \neq \perp$ then
        for $i \leftarrow 0$ to $Z - 1$ do
            if $bucket[l].block[i].bid = \perp$ then
                $bucket[l].block[i] \leftarrow to\_write$
                break
            end
        end
    end
end
```

a schedule is required in order to keep the stash size small, however it has the drawback that the evicted paths may be different from the one fetched to retrieve a certain block. Since this block must be replaced by a dummy block in the ORAM tree, then the client has to re-encrypt the whole path in order to conceal which particular record was extracted. In conclusion, the Access procedure has to fetch and write back three paths, the one where the requested block is located and the two which must be evicted according to the deterministic schedule based on reverse lexicographical ordering. This is a great disadvantage in a remote protocol, since this implementation is the most bandwidth hungry of all. However, Circuit ORAM is designed to be implemented in hardware: even if the "client" is a separate entity, it resides on the same die as the "server", which is a simple memory. The bandwidth cost is not a problem in this local scenario, where data does not travel through a high latency network when exchanged between the client and the server.

Despite the bandwidth, the multiple evictions of Circuit ORAM allows the stash size to be the smallest of all the described protocols: the stash overflow probability decreases as $14 \cdot e^{-S}$, which allows to allocate a stash of a handful of elements even for high security parameters. The theoretical bound still hold when adopting smaller values for Z, which makes Circuit ORAM the solution that incurs in the smallest memory overhead.

2.2.4 Recursive position map

All of the ORAM protocols that we described assume a client that has strongly constrained resources in terms of memory. Even if we focused our attention on the sizing of the stash in order to prevent its overflow, the client needs to store locally the position map as well. It turns out that its size grows linearly with the number of blocks that are stored in the ORAM, making it worthless in case its block size B is in the same order of magnitude of the leaf ids or the number of blocks is too high.

In such cases, the solution is to store the position map recursively into a hierarchy of ORAMs until it requires $\mathcal{O}(1)$ local client storage [33]. Let C be the the number of entries in the local position map of client and let's assume that the blocks stored in the recursive position map host C entries as well. Even though this is not strictly necessary, it simplifies both the complexity analysis and the implementation. Let N be the number of elements contained in the ORAM, with every block id $bid \in [0, N-1]$. Since the ORAMs we presented have the shape of a binary tree with $L = \lceil log_2 N \rceil$ levels and $2^L - 1$ buckets, and can hold up to 2^L blocks, without loss of generality we assume that N is a power of two as well.

The recursive position map consists of the array pos_map with C entries stored on client side and several intermediate ORAM_i, with $1 \leq i \leq R = \lceil \log_C(N) \rceil - 1$, stored on server side. After accessing these data structures, the client retrieves the leaf id of the block with id bid it wants to fetch. The ORAM that actually contains the data is named ORAM_0.

Each leaf id in $pos\_map[j]$ references a portion of the original position map of size:

$$len_0 = \left\lceil \frac{N}{C} \right\rceil$$

that contains indices for block ids in the range $[j \cdot len_0, (j+1) \cdot len_0 - 1]$. Given a block id bid that the client wants to retrieve, the integer $idx_0 = \lfloor \frac{bid}{len_0} \rfloor$ is used to access the locally stored position map pos_map. The leaf id $pos\_map[idx_0]$ is used to fetch a block in ORAM_1 which stores an array pos_map_1 with C entries. The j-th entry of this array references

$$len_1 = \left\lceil \frac{len_0}{C} \right\rceil$$

contiguous block ids in the range $[idx_0 \cdot len_0 + j \cdot len_1, idx_0 \cdot len_0 + (j+1) \cdot len_1 - 1]$. Among these C entries, the client selects the idx_1-th one, where $idx_1 = \lfloor \frac{bid - idx_0 \cdot len_0}{len_1} \rfloor$, which contains the leaf id for a block in ORAM_2.

Eventually, after R recursive steps, the client obtains a block which contains C entries, each referencing $len_R = 1$ block id. The client selects the idx_R-th entry of this block, where $idx_R = \lfloor \frac{bid - \sum_{z=0}^{R-1} idx_z \cdot len_z}{len_R} \rfloor$, obtaining the leaf id of the block with id bid, which can be used to finally fetch the block from ORAM_0.

In order to identify the proper block throughout the Access procedure in each $\mathsf{ORAM}_i, 1 \leq i \leq R$, the client needs to employ unique block ids for each of these ORAMs. Specifically, for ORAM_i, each block is identified by the concatenation of all the indexes $idx_0, \ldots, idx_{i-1}$ employed in all the previous recursion steps to identify the path to be fetched in the next ORAMs. For instance, the block with index idx_0 in ORAM_1 stores the position map of blocks of ORAM_0 with ids in the range $[idx_0 \cdot len_0, \dots, (idx_0+1) \cdot len_0 - 1]$; similarly, the block with index $idx_1||idx_0$ in ORAM_2 stores the position map for blocks of ORAM_0 with ids in the range $[idx_0 \cdot len_0 + idx_1 \cdot len_1, \dots, idx_0 \cdot len_0 + (idx_1+1) \cdot len_1 - 1]$.

The last problem to be addressed is that evictions of fetched blocks in ORAM_i map them to new leaves, thus requiring to update the the block of ORAM_{i-1} which contains the leaf id of the fetched block. To this extent, the new leaf id for the block which will be fetched in ORAM_i can be chosen before the block fetched in ORAM_{i-1} is written back, hence storing in ORAM_{i-1} the updated leaf id for the corresponding block of ORAM_i. Algorithm 2.2.9 outlines the retrieval of the leaf id *lid* for a block with id *bid* in the ORAM_0, combining all the ideas described so far. Two procedures READ and WRITE are employed to fetch and write back elements from the various ORAMs: the former fetches from ORAM_i the path with id *lid* and look-for the block with id *rec_bid* in this path, while the latter evicts the fetched path to ORAM_i by taking into account that the updated leaf id for the block with id *rec_bid* is the value *ev_lid* sampled in the previous iteration.

The retrieval of the correct leaf id via recursion impacts the performance of the ORAM. The time complexity can be evaluated in terms of exchanged data, that is $\mathcal{O}(B \cdot \log(N))$ for any tree-based ORAM if we consider Z as a constant factor which does not depend on the number of elements N. Since the size of ORAM_i is C^i, the resulting time complexity is:

$$\begin{aligned}\sum_{i=1}^{R} B \cdot \log_2(N_i) = \sum_{i=1}^{\log_C(N)-1} B \cdot \log_2(C^i) = B \cdot \log_2(C) \sum_{i=1}^{\log_C(N)-1} i \\ = \mathcal{O}(B \cdot \log_2(C) \cdot \log_C^2(N))\end{aligned}$$

If we change the base of the logarithm, we obtain:

$$\mathcal{O}\left(B \cdot \log_2(C) \cdot \log_C^2(N)) = \mathcal{O}(B \cdot \log_2(C) \cdot \frac{\log_2^2(N)}{\log_2^2(C)}\right) = \mathcal{O}\left(\frac{B}{\log_2 C} \log_2^2 N\right)$$

The number of bits to to encode a leaf id is $\log_2 N - 1$ since there are 2^{L-1} leaves, with $L = \lceil \log_2 N \rceil$. In order to achieve $\mathcal{O}(1)$ storage on the client side, the block size of the recursive position map has to be $B_R \in \mathcal{O}(\log_2 N)$.

Algorithm 2.2.9: RecursiveORAM leaf id resolution

```
Input: A block id $bid$
Output: The leaf id of the block
$len \leftarrow \lceil N/C \rceil$
$rec\_bid \leftarrow \varepsilon$
$offset \leftarrow 0$
$ev\_lid \leftarrow$ GenRandLeaf()
// scan over the client-side pos_map
$idx \leftarrow \lfloor bid/len \rfloor$
$lid \leftarrow pos\_map[idx]$
$offset \leftarrow idx \cdot len$
$rec\_bid \leftarrow idx \,||\, rec\_bid$
$pos\_map[idx] \leftarrow ev\_lid$
for $i \leftarrow 1$ to $R$ do
    // fetch recursive position map chunk
    $rec\_pos\_map \leftarrow \mathsf{ORAM}_i.$Read$(rec\_bid, lid)$
    // find next leaf id
    $len \leftarrow \lceil len/C \rceil$
    $idx \leftarrow \lfloor (bid - offset)/len \rfloor$
    $lid' \leftarrow rec\_pos\_map[idx]$
    // update and write back recursive position map chunk
    $ev\_lid' \leftarrow$ GenRandLeaf()
    $rec\_pos\_map[idx] \leftarrow ev\_lid'$
    $\mathsf{ORAM}_i.$Write$(rec\_bid, lid, rec\_pos\_map, ev\_lid)$
    // update recursive block id and offset
    $rec\_bid \leftarrow idx \,||\, rec\_bid$
    $offset \leftarrow offset + idx \cdot len$
    $ev\_lid \leftarrow ev\_lid'$
    $lid \leftarrow lid'$
end
return $lid$
```

The block size of the last level ORAM B_0 is an independent parameter, and can thus be chosen to minimize the bandwidth due to recursion.

If we choose $B_0 = \log_2 N$, the overhead due to recursion is:

$$\frac{B_R \cdot \log_2^2 N}{B_0} = \frac{\log_2^3 N}{\log_2 N} = \log_2^2 N$$

while choosing a larger block $B_0 = \log_2^2 N$ incurs in a overhead of $\mathcal{O}(\log_2 N)$ [36]. Even if this seems a convenient choice of parameters, such a big block size may become impractical also for moderate ORAM sizes and may be hard to adopt in common applications, that usually require to store much smaller records.

2.2.5 Oblivious Data Structures

The ORAM protocols we have described require a position map, either stored on the client or recursively stored in hierarchical ORAMs of increasing size, in order to access data blocks. The position map allows to access a generic elements in the ORAM. This comes in hand when ORAMs are used to mimic the functionality of a RAM or wrap data structures such as arrays, that in fact can be accessed at random locations. However, several data structures inherently exhibit a hierarchical access pattern, which requires to necessarily retrieve some elements before fetching the desired one. It is the case of trees, whose exploration always starts from the root and follows a path towards a leaf. Each internal node is connected to a limited number of successors, based on the branching factor.

In such cases, random access is not useful, hence the usage of a full position map can be avoided. Oblivious Data Structure (ODS) improve the performance of ORAM schemes leveraging the access pattern of the data structure they encapsulate [41]. For example, an ODS that wraps a tree-like structure requires the client to store only the position map entry that points to the root, since each access to internal nodes will start from the root anyways. The root, in turn, will contain the necessary information to reach its children. Since each ORAM block is uniquely identified by the tuple $(block\_id, leaf\_id)$, it serves as a pointer for the block. Thus, an ORAM block that represents the root of a k-ary tree will hold the node content and k pointers to its children, specified as $(block\_id, leaf\_id)$ tuples. The

Search key	
LEFT CHILD	RIGHT CHILD
Block id	Block id
Leaf id	Leaf id

Figure 2.4: Block of an ODS implementing a binary search tree

structure of internal nodes is depicted in Figure 2.4. While the block id refers to the logical index of an entry, the leaf id tells the client in which path the block is stored. Since the leaf ids are now stored in other blocks of the same ORAM, when these ids are changed then the pointer in the parent node must be updated accordingly. To this extent, the same eviction strategy discussed for the recursive position map, outlined in Algorithm 2.2.9, can be employed. In fact, ODS's can be seen as a sharper way to store the position map recursively.

We will extensively use ODS to wrap binary search trees to implement our algorithms. There are other examples of data structures can benefit from this approach, though, such as graphs [41] and linked lists.

2.3 Substring Search Algorithms

Substring search algorithms address the problem of finding the positions of the repetitions, or *occurrences*, of a string, named *pattern*, in a larger string named *text*. There are many solutions to this problem that can be organized in several taxonomies, according to the size of the inputs, number of patterns to search, matching strategy or additional requirements, such as approximate matching [21].

Here we discuss algorithms that exactly match the pattern and that perform the search in sublinear time with respect to the length of the text. The scenario addressed by these algorithms is the search of many patterns in the same large string, which justifies a computationally intensive preprocessing stage that builds an index of the text. Such index is later used to query as many patterns as needed without applying any change to it. This is usually the only practical solution when searching for many different patterns in a big text, whose size is in the order of hundreds of MB or even GB. On the other hand, *full-text indices* don't usually support updates and require to re-run a full preprocessing step each time the text changes. Since this is a computationally demanding task, adopting a full-text index may be worth only in case the text rarely changes (e.g., when the text is genomic data).

We now introduce some notation that will come in hand in the following sections. A string is a sequence of symbols that belong to a finite alphabet Σ. Its characters exhibit a total order, meaning that it is possible to lexicographically sort strings over Σ. A special symbol $\$ \notin \Sigma$ is optionally used to delimit the end of a string. The total order relationship on Σ is trivially extended to $\Sigma \cup \{\$\}$ by introducing the relationship that the symbol \$ is smaller than any symbol $\sigma_i \in \Sigma$. The characters of the string are indexed in an array-like manner: given a string S, $S[i]$ is its $i+1$-th character, with $i \in [0, |S|-1]$. Substrings are defined via their range: $S[i,j]$ is the substring that begins at the i-th character and ends at the j-th (inclusive). We denote the substring $s[i, |S|-1]$, i.e., a suffix of S, as $S[i,]$, while similarly we denote as $S[,j]$ the prefix of S ending at j (inclusive) (i.e., the substring $S[0,j]$). Given two strings S_1, S_2, $S_1 \sqsubseteq S_2$ denotes that S_1 is a prefix of S_2. By convention, we use P to denote the pattern and T for the text, with their respective lengths $m = |P|$ and $n = |T|$. All the algorithms discussed in the following aims at finding all the positions of occurrences of P in T, that is the set of integers $Pos_{P,T} = \{i \in \{0, \ldots, n-m\} \mid T[i, i+m-1] = P\}$. We will denote the empty string by ε.

2.3.1 Suffix Trees

A suffix tree is a full-text index that provides support for many string processing algorithms, and notably allows to search for a pattern in $\mathcal{O}(m)$ time irrespective of the length of the text. The most efficient algorithm [37] to build a suffix tree runs in $\Theta(n)$ time and requires $\Theta(n)$ space. Even though the asymptotic bounds look promising, they hide very high multiplicative constants: suffix trees occupies about 20x the space of the initial string, making them very demanding in terms of memory consumption. To simplify the construction of suffix trees, we first show how to construct a *suffix trie*, and then we discuss how to obtain a suffix tree from the constructed trie.

A suffix trie is a $|\Sigma|+1$-ary tree which stores all the suffixes of a string

S. Each internal node of a suffix trie may have up to $|\Sigma|+1$ children, and its edges are labeled with a symbol of the alphabet or the end-of-string delimiter \$. Each leaf of the suffix trie corresponds to a suffix of the string S, which can be obtained by concatenating all the labels of the edges along the path from the root to the leaf; thus, each leaf is marked with the index where the corresponding suffix starts in the original string. Specifically, a suffix trie is built by inserting all the suffixes of the string S in the trie, as shown in Algorithm 2.3.2. Algorithm 2.3.1 implements the insertion of a string in the trie. The process starts at the root of the trie: the algorithm consumes one character of the string at every step, either traversing an existing edge marked with the current character of the string, or allocating a new node, linked to its parent with an edge whose label is the current character. The process ends with the creation of a leaf, that stores the current index where the suffix occurs in the original string.

Algorithm 2.3.1: Suffix trie prefix INSERT

```
   Input: The root of a suffix trie
   Input: A string $S$ over the alphabet $\Sigma$
   Input: The index that identifies the string
1  $node \leftarrow root$
2  for $i \leftarrow 0$ to $|S|-1$ do
3      $next\_node \leftarrow node.child[S[i]]$
4      if $next\_node = \perp$ then
5          $next\_node \leftarrow$ CreateEmptyNode()
6          $node.child[S[i]] \leftarrow next\_node$
7      end
8      $node \leftarrow next\_node$
9  end
   // add the leaf $ and mark it with index
10 $node.child[\$] \leftarrow$ CreateLeaf($index$)
```

Although Algorithm 2.3.2 builds a suffix trie in $\mathcal{O}(n^2)$ time, it is simple enough to give the general idea. A more advanced algorithm that exploits a richer representation of internal nodes achieves the same goal in $\mathcal{O}(n)$. The outcome of a run for the string *alfalfa* is shown in Figure 2.5. A suffix tree[1]

[1]even though suffix trees store more information, we neglect it as it doesn't come in hand in substring search

Algorithm 2.3.2: Suffix trie CONSTRUCTION

```
Input: An input string $T$ over an alphabet $\Sigma$
Output: The corresponding suffix trie
1 $trie \leftarrow$ CREATEEMPTYNODE()
2 for $i \leftarrow 0$ to $|T| - 1$ do
3     INSERT($trie, T[i, ], i$)
4 end
  // add the empty string
5 INSERT($trie, \varepsilon, |T|$)
6 return $trie$
```

is a compact form of suffix trie, as it is obtained by erasing all the sequences of nodes with a single edge (e.g., the paths highlighted in red in Figure 2.5) and concatenating their labels in a single one. It is possible to prove that this simple operation allows to shrink the memory required to store a suffix tree to $\mathcal{O}(n)$. The result of edge compression is depicted in Figure 2.6.

It is possible to immediately spot some recurring properties of suffix trees. By construction, they have $|S| + 1$ leaves, each one corresponding to one of the suffixes of S or the empty string ε. The number of internal nodes is not fixed, but it is upper bounded by $2 \cdot |S|$. The branching factor is at most $(|\Sigma| + 1)$, as, for all the edges starting on the same node, their labels start with distinct characters; however, each internal node has a varying number of children, according to the structure of the initial string. Moreover, both the depth and the number of nodes at the same level highly depend on the structure of the original string, and is not known in advance. Given a path from the root of the tree to a node, we call the concatenation of all the edge labels along this path as *path label*. A path label can lead to only one internal node by construction.

In order to employ suffix tree to look for occurrences of a pattern P, it is worth noting that each occurrence of P is a prefix of a suffix: indeed, as an occurrence of P is defined as an index $i \in \{0, \dots, n - m\}$ such that $T[i, i+m-1] = P$, then P is necessarily a prefix of $T[i,]$. Therefore, all the occurrences of a pattern can be retrieved via a simple tree exploration, since a suffix tree allows to easily find all the prefixes of the suffixes T. Algorithm 2.3.3 describes this process. The exploration starts at the root of the suffix

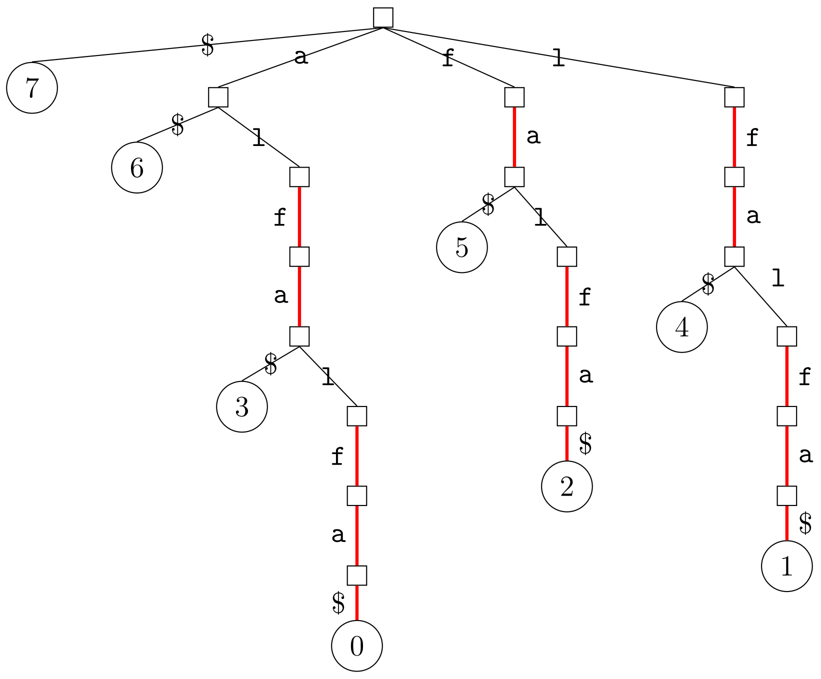

Figure 2.5: Suffix **trie** for the string *alfalfa*

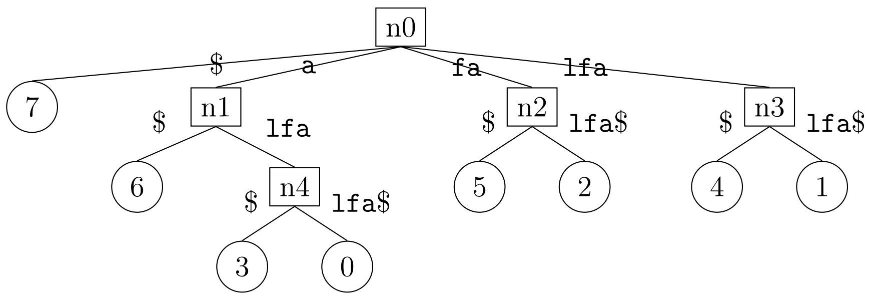

Figure 2.6: Suffix **tree** of the string *alfalfa*

tree, looking for the edge whose label starts with the first character of the pattern (line 5. If there is no such edge, the search for a substring fails, otherwise the edge label *label* is compared against the string. If the label is a prefix of the current pattern, $|label|$ characters of the pattern are consumed, the edge is traversed and the process restarts at the next node, considering the suffix of the pattern that starts at index $|label|$. If a mismatch occurs in the middle of *label*, then the pattern does not occur in the text. The process goes on until the whole pattern is consumed: if this happens in the middle of an edge, it is traversed anyways. The sequence of nodes explored by a pattern P is fixed since each internal node has at most one child whose edge label begins with a specific $\sigma_i \in \Sigma$. Moreover, all the suffixes beginning with P will traverse the same exact path, eventually passing through in the final node reached by P. Hence, all the leaves of the subtree rooted at that node represent suffixes whose prefix is P, hence their tags represent the positions of the text where the pattern occurs.

For example, when looking for the pattern $P =$ `lfal`, the algorithm first descends in the intermediate node `n3`, consuming the first three characters of P. Then it goes to the leaf marked with `1`, that is, in fact, the index where *lfal* occurs in *alfalfa*.

2.3.2 Suffix Arrays

Suffix arrays hold the starting indices of all the suffixes of a string once they are sorted lexicographically. Although less powerful than suffix trees, they represent a much more compact full-text index. In fact, they contain exactly $n + 1$ entries, each one requiring $\lceil \log_2 n \rceil$ bits to store the position where a suffix starts. The overhead in terms of memory consumption in order to store a suffix array is $\lceil \log_2(n) \rceil / \lceil \log_2(|\Sigma|) \rceil$, as $\lceil \log_2(|\Sigma|) \rceil$ is the number of bits used to encode the characters of the alphabet Σ.

A trivial strategy to generate a suffix array is applying a sorting algorithm to the suffixes of a string. The complexity of sorting is $\mathcal{O}(n \log n)$, while the lexicographical comparison between strings of length between 0 and n takes at most $\mathcal{O}(n)$, yielding a total complexity of $\mathcal{O}(n^2 \log n)$. However, there is an efficient algorithm that exploits the natural redundancy of the suffixes of the string to perform the same task in $\Theta(n)$ time and space [26].

A more original way to achieve the same purpose is lexicographically

Algorithm 2.3.3: Suffix tree SubstringSearch

```
Input: An input string $P$ over an alphabet $\Sigma$
Input: The root node of a suffix tree $stree$
Output: The occurrences of $P$ in the tree

$node \leftarrow stree$
$i \leftarrow 0$
while $i < |P|$ do
    $leading\_char \leftarrow P[i]$
    $node \leftarrow node.child[leading\_char]$
    if $node = \perp$ then
        // a mismatch occurred
        return $\varnothing$
    end
    else
        /* we assume that the edge label is stored in the
           child node                                      */
        $label \leftarrow node.label$
        if $label \sqsubseteq P[i,]$ then
            // match |label| characters of the pattern
            $i \leftarrow i + |label|$
        end
        else if $P[i,] \sqsubseteq label$ then
            // all the pattern has been consumed
            $i \leftarrow |P|$
        else
            return $\varnothing$
        end
    end
end
return all the leaf tags of the subtree rooted at node
```

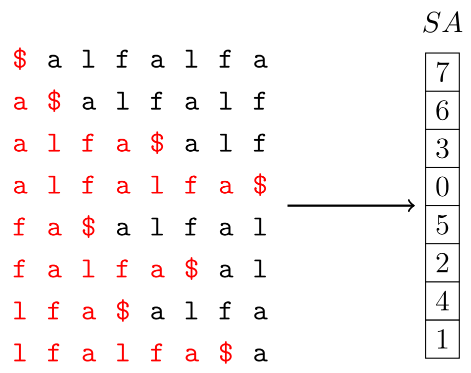

Figure 2.7: Suffix **array** SA of the string *alfalfa*

sorting the outgoing edges of each node in the suffix tree, placing as the leftmost edge the one with the lexicographically smallest label and as the rightmost edge the one with the lexicographically greatest label, and then visit the tree with a depth-first search, merging into an array all the ids of the visited leaves (which corresponds to the indexes of a suffix by construction of the suffix tree). The suffix array for the string *alfalfa* is shown in Figure 2.7.

Each row of the matrix on the left side is a rotation of the initial string *alfalfa$*. The characters highlighted in red represent the suffix up to the terminator. As all the rows are lexicographically sorted, then the suffixes are lexicographically sorted too; the suffix array is built by concatenating the indices of the suffixes of the original string, i.e., the starting positions of the suffix in the original string. It is easy to note the order of the indexes is the same as the leaf ids of the suffix tree in Figure 2.6.

Looking for occurrences of a pattern P in a suffix array still relies on the property that each occurrence is a prefix of a suffix. In particular, as the suffix array contains the indices of all the suffixes in lexicographical order, then all the suffixes with the same prefix P represents a contiguous portion in the suffix array: therefore, to identify this portion, it is sufficient to identify the lexicographically smallest an greatest suffix whose prefix is P. As the suffixes are lexicographically sorted in the array, these can be efficiently retrieved with two binary searches. Algorithm 2.3.4 shows the binary search to find the least suffix. Its only peculiarity is that whenever a suffix starting with P is found, the search continues in the left subtree in order to potentially find a lexicographically smaller suffix (line 9); in the meanwhile, *idx* keeps the index of the smallest occurrence happened so far. Since the search continues in the left subtree when there is a match, this algorithm is informally named *leftmost* binary search: the rightmost search is trivially its dual, as it proceeds in the right subtree to find a greater match. A binary search on an array with $n+1$ requires $\mathcal{O}(\log n)$ iterations; in our case, at each iteration the lexicographic comparison between the pattern P and the suffix is performed, whose cost is $\mathcal{O}(m)$, thus yielding an overall complexity $\mathcal{O}(m \log n)$.

Algorithm 2.3.4: Suffix array LeftmostBinarySearch

```
   Input: An input string $P$ over an alphabet $\Sigma$
   Input: The suffix array $SA$
   Output: The leftmost index of the $SA$ entry prefixed by $P$
 1 $idx \leftarrow -1$
 2 $start \leftarrow 0$
 3 $end \leftarrow |SA| - 1$
 4 while $start \leq end$ do
 5     $middle \leftarrow start + \lfloor \frac{end-start}{2} \rfloor$
 6     suffix $\leftarrow T[SA[middle],]$
       /* if the pattern is a prefix of that suffix, update
          index                                              */
 7     if $P \sqsubseteq$ suffix then
 8         $idx \leftarrow middle$
           /* if there is a match, go to the left in order to
              find another leftmost match                    */
 9         $end \leftarrow middle - 1$
10     end
11     else
           // go left, common dichotomic search
12         if $P >$ suffix then
13             $end \leftarrow middle - 1$
14         end
15         else
16             $start \leftarrow middle + 1$
17         end
18     end
19 end
20 return $idx$
```

```
a l f a l f a $
$ a l f a l f a
a $ a l f a l f
f a $ a l f a l
l f a $ a l f a
a l f a $ a l f
f a l f a $ a l
l f a l f a $ a
```

Figure 2.8: First step of the BWT for the string *alfalfa*

In order to avoid the comparison of the whole pattern every time, suffix arrays can be enriched with another array, the Longest Common Prefix (LCP) array, that stores how long the common prefix between any entry and its left/right child is. The complexity of the search drops to $\mathcal{O}(m + \log n)$ if matching characters in the LCP are skipped.

2.3.3 Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) of a string is a permutation of its characters, plus the delimiter \$, that exhibits remarkable features. Given a text T, it is obtained applying the following transformations [4]:

1. generate all the right (left) rotations of a string and arrange them in a matrix, obtaining $n + 1$ rows with $n + 1$ elements;

2. sort the rows lexicographically, obtaining the *BWT matrix*;

3. The characters in the last column constitutes the BWT of the original string, denoted by L

If we consider our running example *alfalfa*, after the first step we obtain the matrix in Figure 2.8, where the initial character of the string is in evidence. After the second, the outcome is exactly the same as in Figure 2.7. Hence, the BWT of our running example will be $L = aff\$llaa$.

We will now present some of the properties of the BWT. Let us consider the first and last column of the BWT matrix, as depicted in Figure 2.7. They will be `$aaaffll` and `aff$llaa` respectively. To this extent, we introduce some additional notation which will be useful throughout the discussion of the properties. Given a string S over Σ and the delimiter \$, we denote as $pos_S(\sigma)$ the position in S of the character $\sigma \in \Sigma \cup \{\$\}$.

Lemma 1. *Let L be the last and F the first columns of the BWT matrix built for string T. For each pair of corresponding characters in F and L, i.e., $F[i]$ and $L[i]$, $i \in \{0, \ldots, n\}$, $pos_T(F[i]) = pos_T(L[i]) + 1 \bmod (n+1)$, which means that $L[i]$ preceeds $F[i]$ in the original string T.*

Proof. The proof is trivial. In fact, each row of the BWT matrix is, by construction, a rotation of the original string. The delimiter \$ preceeds the first character $T[0]$ in each row of the matrix as shown in Figure 2.8. This property is preserved when permutating the rows, and hence stands for the index k of L where \$ occurs, as well. □

Lemma 1 allows to explain how the BWT can be efficiently constructed from the suffix array. Indeed, the suffix array stores the position in the original string of $F[i]$, while the i-th character of the BWT is $L[i]$, which precedes $F[i]$ and thus occurs at position $SA[i]-1$. Therefore, the BWT of a string T can be constructed in $\mathcal{O}(n)$ time from the suffix array as $BWT_T[i] = T[SA[i] - 1 \bmod (n+1)]$.

Lemma 2. *Let T be a string over the alphabet Σ. For each pair of indices $k_1, k_2 \in \{0, \ldots, n\}$ where $T[k_1] = T[k_2]$, it holds that:*

$$pos_F(T[k_1]) < pos_F(T[k_2]) \Leftrightarrow pos_L(T[k_1]) < pos_L(T[k_2])$$

Proof. Suppose that $pos_F(T[k_1]) < pos_F(T[k_2])$. As the string F contains the starting character of the suffixes in lexicographical order, than this means that $T[k_1,] < T[k_2,]$. As $T[k_1] = T[k_2]$, this necessarily implies that $T[k_1+1,] < T[k_2+1,]$, therefore $pos_F(T[k_1+1]) < pos_F(T[k_2+1])$, since F is constructed from sorted suffixes. Denote $f_1 = pos_F(T[k_1+1])$ and $f_2 = pos_F(T[k_2+1])$. By Lemma 1, then $L[f_1] = T[k_1]$ and $L[f_2] = T[k_2]$, therefore $pos_L(T[k_1]) = f_1$ and $pos_L(T[k_2]) = f_2$, which implies that $pos_L(T[k_1]) < pos_L(T[k_2])$.

Conversely, suppose that $pos_L(T[k_1]) < pos_L(T[k_2]$. By Lemma 1, $pos_F(T[k_1+1]) = pos_L(T[k_1])$ and $pos_F(T[k_2+1]) = pos_L(T[k_2])$, thus $pos_F(T[k_1+1]) < pos_F(T[k_2+1])$, which necessarily implies that the suffix $T[k_1+1,]$ is smaller

\$	a	l_1	f	a	l_2	f	a
a	\$	a	l_1	f	a	l_2	f
a	l_2	f	a	\$	a	l_1	f
a	l_1	f	a	l_2	f	a	\$
f	a	\$	a	l_1	f	a	l_2
f	a	l_2	f	a	\$	a	l_1
l_2	f	a	\$	a	l_1	f	a
l_1	f	a	l_2	f	a	\$	a

Figure 2.9: Visualization of Lemma 2

than the suffix $T[k_2+1,]$. If we now consider the suffixes $T[k_1,]$ and $T[k_2,]$, then since $T[k_1] = T[k_2]$, then $T[k_1+1,] < T[k_2+1,] \Rightarrow T[k_1,] < T[k_2,]$. Therefore, since the string F contains the starting characters of the suffixes in lexicographical order, then $pos_F(T[k_1]) < pos_F(T[k_2])$ $\square$

Lemma 2 basically claims that, for each character $\sigma \in \Sigma$, if we enumerate all the occurrences of σ in T, then these occurrences appear in the same order in both F and L. For instance, in Figure 2.9, the letter l has been enumerated, obtaining the string *al₁fal₂fa\$*. It's easy to observe that *l_2* preceeds *l_1* both in the string F and in the string L.

A third lemma can be easily derived from Lemma 2. This lemma employs the concept of *rank*. Let S be a string over an alphabet Σ and $\sigma \in \Sigma$ a symbol of the alphabet. We define the rank as:

$$rank_{S,\sigma}(x) = |\ \{i \in [0,x) \mid S[i] = \sigma\}\ |$$

The rank of a character σ up to position x is basically the number of occurrences of σ in $S[0, x-1]$.

Lemma 3. *Let T be a string over the alphabet Σ. For each character $T[k]$, $k \in \{0,\dots,n\}$ in T, define $x_i = pos_F(T[k])$ and $y_i = pos_L(T[k])$. Then, for all $k \in \{0,\dots,n\}$, $rank_{F,T[k]}(x_i) = rank_{L,T[k]}(y_i)$.*

Proof. If $rank_{F,T[k]}(x_i) = r$, then it means that there are r occurrences of $T[k]$ in $F[0, x_i - 1]$, denoted by $T[k_1],\dots,T[k_r]$. Suppose, without loss of generality, that $pos_F(T[k_1]) < \dots < pos_F(T[k_r]) < x_i$. As $T[k_1] =$

$\cdots = T[k_r] = T[k]$, Lemma 2 is applicable, therefore $pos_L(T[k_1]) < \cdots < pos_L(T[k_r]) < y_i$, which means that $rank_{L,T[k]}(y_i) = r = rank_{F,T[k]}(x_i)$. $\square$

F and L contain enough information about the original text to implement an efficient string searching algorithm [15]. The column L is the BWT of the original string, while the information in F can be easily compressed in $\mathcal{O}(|\Sigma|)$ space instead of $\mathcal{O}(n)$. All the occurrences of a symbol $\sigma \in \Sigma$ are contiguous in F, so it's possible to represent this information via a dictionary $\mathcal{C}$ that tracks the first position in F where each character σ occurs. In the running example of Figure 2.7, $\mathcal{C}[\$] = 0$, $\mathcal{C}[a] = 1$, $\mathcal{C}[f] = 4$ and $\mathcal{C}[l] = 6$. For convenience, we also define another entry $\mathcal{C}[*] = |T| + 1$ that in our example is $\mathcal{C}[*] = 8$. From there on, we will assume the existence of a function $next(\sigma_i) = \sigma_{i+1}$, that maps each character of the alphabet to the next one in alphabetical order. The last character of the alphabet will be mapped to $*$ instead.

The string matching algorithm that we will describe is known as *backwards search*, as it starts to match the pattern from the last character. Given a pattern P of length m, it finds the occurrences of $\{P[m-1], P[m-2, m-1], \ldots, P[m-i, m-1], \ldots, P[0, m-1]\}$ by successive refinements. For example, when looking for *lfal*, it matches the sequence of patterns $\{l, al, fal, lfal\}$. These occurrences are identified by a range of contiguous characters in F, which represents all the starting characters of suffixes which are prefixed by an increasing portion of the pattern; specifically, in the i-th iteration, the occurrences of $P[m-i, m-1]$ are represented by two indexes $start_i$, end_i, which identifies the substring $F[start_{m-i}, end_{m-i}]$ composed by the starting characters of all the suffix prefixed by $P[m-i, m-1]$. In the first iteration, the starting and ending index that match the occurrences of $P[m-1]$ in T are trivially retrieved from the $\mathcal{C}$ dictionary:

$$\begin{cases} start_{m-1} = \mathcal{C}[P[m-1]] \\ end_{m-1} = \mathcal{C}[next(P[m-1])] - 1 \end{cases}$$

and the total number of occurrences is given by $end_{m-1} - start_{m-1} + 1$. Now suppose to add the character placed at index $m-2$. The algorithm aims at finding two indexes $start_{m-2}$ and end_{m-2} representing all the occurrences of $P[m-2, m-1]$ in T. For sure, $start_{m-2}$ and end_{m-2} will both fall in the interval defined by $[\mathcal{C}[P[m-2]], \mathcal{C}[next(P[m-2])]-1]$. However, only the suffixes starting with $P[m-2]$ that are followed by $P[m-1]$ represent a match and must be included in the new range. We now show how to retrieve them.

The expression $rank_{L,P[m-2]}(end_{m-1}+1) - rank_{L,P[m-2]}(start_{m-1})$ returns the number of occurrences of $P[m-2]$ in $L[start_{m-1}, end_{m-1}]$. By Lemma 1, they are followed by $P[m-1]$ in the original string, since $P[m-1]$ is the corresponding character in the F column. Hence, they also represent matches for the last two characters of the pattern, and the difference between the two ranks is equivalent to $end_{m-2} - start_{m-2} + 1$. Let $y_1, y_2 \in [start_{m-1}, end_{m-1}]$ be the first and last positions of $L[start_{m-1}, end_{m-1}]$ where the character $P[m-2]$ occurs. Of course, $rank_{L,P[m-2]}(end_{m-1} + 1) = rank_{L,P[m-2]}(y_2+1)$, as no characters equal to $P[m-2]$ occur in $L[y_2+1, end_{m-1}]$, and similarly $rank_{L,P[m-2]}(start_{m-1}) = rank_{L,P[m-2]}(y_1)$, since no characters equal to $P[m-2]$ occur in $L[start_{m-1}, y_1-1]$. Denote the positions in the original string T of the characters $L[y_1]$ and $L[y_2]$ as, respectively, k_1 and k_2, i.e., $k_1 = pos_T(L[y_1])$ and $k_2 = pos_T(L[y_2])$ and define as $x_1 = pos_F(T[k_1])$ and $x_2 = pos_F(T[k_2])$. In particular, as all the occurrences of a symbol $\sigma \in \Sigma$ are contiguous in F, then x_1 is obtained as the sum of the position of the first occurrence of $P[m-2]$ in F (i.e., $\mathcal{C}[P[m-2]]$) to the number of occurrences of $P[m-2]$ preceding $T[k_1]$ in F (i.e., $rank_{F,P[m-2]}(x_1)$); similarly, x_2 is obtained as the sum of the position of the first occurrence of $P[m-2]$ in F (i.e., $\mathcal{C}[P[m-2]]$) to the number of occurrences of $P[m-2]$ preceding $T[k_2]$ in F (i.e., $rank_{F,P[m-2]}(x_2)$). As by Lemma 3, $rank_{F,P[m-2]}(x_2) = rank_{L,P[m-2]}(y_2)$ and $rank_{F,P[m-2]}(x_1) = rank_{L,P[m-2]}(y_1)$, then we obtain $x_1 = \mathcal{C}[P[m-2]] + rank_{L,P[m-2]}(y_1)$ and $x_2 = \mathcal{C}[P[m-2]] + rank_{L,P[m-2]}(y_2)$. We observe that $rank_{L,P[m-2]}(y_2) = rank_{L,P[m-2]}(y_2+1) - 1$, as $L[y_2] = P[m-2]$, therefore, in conclusion we obtain that:

$$\begin{aligned}
x_1 &= \mathcal{C}[P[m-2]] + rank_{L,P[m-2]}(y_1) = \\
&= \mathcal{C}[P[m-2]] + rank_{L,P[m-2]}(start_{m-1}) \\
x_2 &= \mathcal{C}[P[m-2]] + rank_{L,P[m-2]}(y_2+1) - 1 = \\
&= \mathcal{C}[P[m-2]] + rank_{L,P[m-2]}(end_{m-1}+1) - 1
\end{aligned}$$

The integers x_1, x_2 identify the range $F[x_1, x_2]$, composed by the starting characters of suffixes which are prefixed by $P[m-2, m-1]$, that are all the occurrences of $P[m-2, m-1]$. Therefore, x_1 and x_2 can be assigned to the new indexes $start_{m-2} = x_1$ and $end_{m-2} = x_2$.

By generalizing this reasoning for all the characters $P[i]$ of the pattern, we obtain:

$$\begin{cases} start_{m-i} = \mathcal{C}[P[m-i]] + rank_{L,P[m-i]}(start_{m-i+1}) \\ end_{m-i} = \mathcal{C}[P[m-i]] + rank_{L,P[m-i]}(end_{m-i+1} + 1) - 1 \end{cases}$$

We observe that, when a pattern is not found in a text, both the ranks in the previous expressions will be equal, and thus $start_{m-i} > end_{m-i}$, effectively leading to a number of occurrences equal to $end_{m-i} - start_{m-i} + 1 = 0$. Algorithm 2.3.5 summarizes this procedure. The indices *start*, *end* computed by Algorithm 2.3.5 identify the range $F[start, end]$ which contains the starting characters of the suffixes of T prefixed by P. The positions of these characters in the original string, which are the occurrences of P, are stored in the corresponding entries of the suffix array, $SA[start, end]$. The computational complexity of Algorithm 2.3.5 clearly depends on the cost of computing *rank* queries. Given the BWT L, $rank_{L,\sigma}(i)$, for $i \in \{0, \dots, n\}$ and $\sigma \in \Sigma$, can be precomputed and stored in a table with $|\Sigma|$ rows and $n+1$ columns; thus, the *rank* queries in Algorithm 2.3.5 can be implemented by simply retrieving elements from this table with $\mathcal{O}(1)$ cost, yielding an overall complexity $\mathcal{O}(m)$ for Algorithm 2.3.5.

2.4 Related Works

The related works are divided in two different categories that are mostly orthogonal. The first one includes works that implement secure substring search protocol only relying on cryptographic constructions. One of the presented solutions adopts techniques recurring in the context of SSE to safely query an index based on suffix tree [7], while an ORAM based solution is presented in [25]. The other line of works focuses on the evaluation of the security of enclaved programs through their leakage profile [17], or put in place specific practices that try to mitigate side channels at their root [24,43],

Algorithm 2.3.5: BackwardsSearch using BWT

```
Input: An input string $P$ over an alphabet $\Sigma$
Input: The array $\mathcal{C}$
Input: The BWT $L$ of a text $T$
Output: The indices of the occurrences
1  $m \leftarrow |P|$
2  $start \leftarrow \mathcal{C}[P[m-1]]$
3  $end \leftarrow \mathcal{C}[next(P[m-1])] - 1$
4  for $i \leftarrow m-2$ to $0$ do
5      $\sigma \leftarrow P[i]$
6      $start \leftarrow \mathcal{C}[\sigma] + rank_{L,\sigma}(start)$
7      $end \leftarrow \mathcal{C}[\sigma] + rank_{L,\sigma}(end+1) - 1$
8      if $start > end$ then
9          break
10     end
11 end
12 return $(start, end)$
```

rather that relying on automated techniques to retrofit existing software. While not directly related to our application scenario, the latter provide useful reference for developing applications that directly address the powerful threat model of Intel SGX.

2.4.1 Private Substring Search Protocols

The algorithm proposed in [7] exploits an alternative representation of suffix trees based on dictionaries (Subsection 2.3.1). Specifically, in a suffix tree, for each internal node the labels of all edges start with a distinct character of the alphabet; thus, the concatenation of the path label of the node and the first character of an edge, referred to as *initial path label*, uniquely identifies all children of the node at hand. As an example, in Figure 2.6, *alfa* is the path label for `n4`, but `al` is sufficient to uniquely identify this node. This property can be used to arrange a suffix tree as a dictionary, that uses a hash function over the initial path label to identify each internal node of the suffix tree. In particular, [7] employs a keyed hash function F_{k_1}, or equivalently a keyed Pseudo-Random Function (PRF), to compute a hash of the initial path label for each node, which is used to index a dictionary.

The dictionary entries contain information about the initial path label v_i of the children, in particular the result of $F_{k_2}(v_i)$. In order to hide information about the initial text, the dictionary is padded to have exactly $2N$ nodes, each with exactly $|\Sigma|$ children in scrambled order. Furthermore, each dictionary entry contains an index that points to the full edge label, as well as the index of the leftmost leaf and the number of leaves in its subtree.

The schemes additionally employs Π, a symmetric cipher offering Authenticated Encryption (AE) and randomization via an IV, that exposes the encryption $\Pi.Enc_{key}(ptx)$ and decryption $\Pi.Dec_{key}(ctx)$ functionalities, returning $\perp$ when the key is wrong.

A client that wants to search for the pattern P in the data structure, uploads a series of tokens $\Pi.Enc_{F_{k_2}(P[0,i])}(F_{k_1}(P[0,i]))$ for $i \in [0, |P| - 1]$, corresponding to a prefix of P.

During the first part of the protocol, finds the longest prefix of the pattern that matches the initial path label of a node. In particular, the server starts from the dictionary entry corresponding to the root node and employs the hashed initial path labels of the children as as a key to decrypt the tokens. When decryption is successful, a token matches the initial path label of one fo the children node, which is chosen as the next node to be explored. The plaintext obtained indexes this node in the dictionary, thus it is then employed to retrieve the dictionary entry corresponding to the selected children. With a similar procedure, the server explores the children of this node to find a node whose initial path label corresponds to one of the tokens. The server iterates this exploration until a child node whose initial path label matches a token is found. Algorithm 2.4.1 shows how tree exploration is performed.

Since this phase is performed by the server, it is able to establish the length of the longest common prefix between subsequent queries by inspecting the sequence of accesses to the dictionary. However, the fact that tokens are encrypted prevents the server from deriving this information a priori, and in particular, in the case two queries don't occur in the original text.

At the end of this phase, the server obtains a node whose initial path label is the longest prefix of the P which matches the initial path label of a node in the suffix tree. In the second phase, the client downloads the full edge label of a node to check whether or not the suffix of the pattern

Algorithm 2.4.1: Suffix tree substring search in SSE setting

```
Input: The dictionary $D$ containing the suffix tree
Input: The sequence of tokens $Tok$ generated from a pattern $P$
Output: The node corresponding to the longest initial path label
        among the tokens

/* the entry corresponding to the root is known on the
   server side                                                  */
$node \leftarrow D[F_{k_1}(\varepsilon)]$
$i \leftarrow 0$
while $i < |P|$ do
    $next\_key \leftarrow \perp$
    $j \leftarrow i$
    while $j < |P| \land next\_key = \perp$ do
        $k \leftarrow 0$
        while $k < |\Sigma| - 1 \land next\_key \neq \perp$ do
            $next\_key \leftarrow \Pi.Dec_{node.child[k]}(Tok[j])$
            if $next\_key \neq \perp$ then
                $i \leftarrow j + 1$
            end
            $k \leftarrow k + 1$
        end
        $j \leftarrow j + 1$
    end
    if $next\_key \neq \perp$ then
        $node \leftarrow D[next\_key]$
    end
    else
        break
    end
end
return $node$
```

is contained in the edge label. If so, P occurs in the text, and the client will also retrieve the indices of the occurrences, by employing the index of the leftmost leaf and the number of leaves of the subtree rooted in the node identified in the first phase, an information stored in the dictionary entry of the node at hand. This index and the number of leaves are employed to access another data structure which allows to retrieve the actual occurrences of the pattern in the text. The characters of the original text are stored in a separate array in permuted order, so that an attacker does not know where an edge label is placed within the original text. This also holds for the leaves of the suffix tree, that would otherwise leak the lexicographical ordering of the suffixes that potentially match a pattern. Nonetheless, the server is still able to determine the similarity of subsequent queries by comparing the sets of accessed locations.

Although the usage of AE allows the client to verify if the server is misbehaving, thus opening to *malicious* server scenarios, the possibility to relate successive queries is a huge drawback of this protocol. We observe that this is possible thanks to the inspection of the logical access pattern of the application, that in this case doesn't require the exploitation of any side channel. With enough domain knowledge, a similar amount of leakage has already been showed to be sufficient to recover significant portions of data [5, 19, 29].

To the extent of totally hiding the correlation between queries, [25] hides the nodes traversed during a suffix tree exploration by employing an ORAM for each level of the tree. This solution is tailored for the case where the original text T is the concatenation of d independent strings whose length l is in the order of hundreds of characters, and in general $d >> l$. Since a substring cannot span multiple strings, the maximum length of a suffix in this tree is l. Suffix trees exhibit an irregular structure: while the maximum number of levels is l, the number of nodes residing on the same level cannot be predicted. In order to overcome this issue, internal nodes are distributed according to specific rules that guarantee that the size of each level is bounded. The figure of merit to evaluate the efficiency of the protocol is its communication cost, since the ORAMs reside on a remote server. The protocol exhibits poly-logarithmic cost with respect to the number of documents: for instance, when the length of each string l is $O(\log(d))$, its communication cost is $\mathcal{O}(\log\log d \cdot \log^3 d)$. Nevertheless, this cost depends linearly from the length l, making this protocol unpractical for long strings.

The same work proposes also a simpler construction based on suffix arrays with cost $\mathcal{O}(\log^3 d)$ which does not linearly depend on the length l of the string. Nonetheless, this solution doesn't allow to update the suffix array with more strings after the data structure has been constructed. A suffix array can be arranged in memory as a k-ary search tree: each internal node contains a pointer for each of the k children; in addition, for each children, the node stores the smallest suffix that belongs to the subtree rooted in the children. The implementation sets $k = 2$: thus, a slightly modified version of Algorithm 2.3.4 and its dual allow to retrieve the occurrences of a pattern. In order to choose the correct child, each node stores k suffixes. A compressed representation avoids to store the suffixes in internal nodes, and defers character-by-character comparisons to the last level of the tree, that needs to store entire suffixes. While this solution works well in the case of multiple concatenated string, this solution is inefficient for a large, unique text, whose suffix lengths are in the same order of magnitude of its size, requiring a lot of space in the last level.

2.4.2 Intel SGX based Applications

Several applications employ Intel SGX to preserve the confidentiality of sensitive data, estimating the amount of information that leak during the execution of an enclave. The security of applications is usually intended relatively to a specific leakage model, that establishes which information is not confidential and can be safely learnt from the untrusted server without compromising confidentiality. While some domains have well established models, in other cases the choice is driven by the security requirement of the application. Thus, even enclaves with a lenient leakage profile are marked as secure if they don't exceed the leakage threshold established by the enclave developer.

HardIDX [17] implements a safe index based on *B+-trees*, a popular data structure in database design that optimizes access to large block devices such as hard drives. B+-trees store actual data in the leaves, while internal nodes serve as k-ary search tree. When the B+-tree fits in the EPC, the search can be performed entirely within the enclave, leaking the access patterns at the granularity of a 4kB page. Nonetheless, a page may contain multiple nodes; furthermore, the enclave explores nodes in the same level of the tree in arbitrary order, thus limiting the amount of information inferred by the server about the topology of the B+-tree. In most cases, however, the limited size of the EPC forces the enclave to move only a subset of the nodes in the reserved memory. Despite the nodes are secretly permuted before being moved outside the enclave, an attacker may formulate hypotheses on the structure of the tree at the granularity of nodes, and observe subsequent queries to both establish their similarity, progressively refining its guesses about topology. The adopted leakage model takes into account this kind of attacks. Notably, HardIDX offers protection against active attackers that don't comply to the protocol. As the confidentiality and integrity of the nodes stored outside the enclave is guaranteed by the usage of AE schemes, the main risk is that an attacker doesn't return all the results to the client, or omits some nodes requested by the enclave during the search phase. Multi-set hashes provide protection against untrustworthy servers: a hash of the ids of requested nodes is accumulated in the enclave and is compared to the hash of ids of the nodes handed in by the untrusted host. If the enclave detects a discrepancy at the end of the search, it aborts the computation. The same procedure is applied on the client side when checking the authenticity of the results. Since the value of a multi-set hash does not depend on the order in which elements are added, they are compatible with the permutation of internal nodes which is employed and results, necessary to partially conceal the topology of the tree. Furthermore, chosen-query attacks are not possible since the queries are encrypted with a key that is shared only between the enclave and the remote client. Any attempt to execute rogue queries results in wrong decryption which can be detected via AE.

Opaque [43] implements oblivious relational operators that allow to perform Spark SQL queries on encrypted data. In fact, the adoption of sole encryption of database entries has proved to be insufficient in providing protection against motivated attackers, who may discover information looking at the access patterns, such as which entries of a table are associated to the same foreign key. In order to provide access pattern obliviousness, Opaque focuses on providing an oblivious sort primitive in a distributed scenario, that is pivotal for implementing database primitives. *Sorting networks*, that have $\mathcal{O}(n \log^2 n)$ complexity, perform a sequence of comparisons and swaps that does not depend on the input data, thus exhibiting an access pattern which is independent from the input data; this property allows to make any information about access pattern inferred from side channels useless when data are obliviously sorted inside a secure enclave. In order to open up to a distributed scenarios, sorting networks are combined with a scrambling procedure which allows to avoid leaking access patterns to adversaries which observe the amount of data exchanged among different nodes. The obtained sorting method is named *column sort*: local data is first sorted within each node through a sorting network, and then shuffled among different machines in a fixed number of rounds, until each node has a portion of the sorted array. Sorting allows to implement in an oblivious manner many relational operators. A `FILTER` is obtained by obliviously sorting entries according to the assigned label, that is 0 if the condition is not met, and 1 otherwise. In order to conceal the number of rows that pass the filter, all of them are carried along the computation and later discarded by the client. Aggregation queries (e.g. `GROUP BY` operator in SQL), which aims at partitioning the records according to an aggregation attribute and then computes an aggregate value for each partition, can be performed by sorting all the rows based on aggregation attributes and evaluating the aggregate value for each partition. Then, the nodes exchange intermediate results to determine the final value. A primary-foreign key `JOIN` can be achieved by the means of sorting as well: the primary key table T_p and the foreign key table T_f are concatenated, and sorted by the join attributes, placing T_p entries before T_f entries in case of ties. Each primary key occurs only once in T_p, ensuring that a single T_p key will be followed by all the entries of T_f that will be joined with it. For each entry, a node emits either a dummy record or a correctly joined one. Oblivious sorting once again allows to discard all the dummies. Intel Architectural Enclave (Intel AE) ensures that the data received by a node is authentic, and generates a MAC that certifies that the output comes from a trustworthy enclave. It is sufficient that all the nodes and the client share an ephemeral symmetric key. The computation starts from the client, that generates the initial tokens for the query, and that later verifies the MAC of the results to ensure that the attacker didn't tamper with the computations.

Oblix [24] implements a sorted multimap, that implements efficient retrieval of elements based on a search *key* even when it is shared among different entries. A possible implementation of sorted multimaps results from overlapping a self-balancing tree, such as an *AVL tree*, that preserves efficiency even after updates, with an *order statistics binary tree*, that keeps track of the number of entries in the left and right subtree of a node that share its key. The main idea of the work is concealing the access pattern exploiting an ORAM whose client resides in the enclave: it exploits an ODS to store the position map recursively, leveraging the tree-like access pattern of the sorted multimap, and operates on obliviously on the stash via linear scans or using sorting networks. While a previous paper showcased this solution [31] highlighting the advantages of Circuit ORAM, Oblix optimizes Path ORAM achieving better results. The protocol was successfully employed to implement the private contact discovery of Signal and Google's Key Transparency, besides a searchable encryption protocol that retrieves all the documents where a certain keyword occurs. The implementation details are here omitted, since our work is based on the same principles.

SSE schemes can also be backed by enclaved execution, as in [1]. The protocol they describe, named *Bunker*, retrieves the documents associated to a certain keyword using an oblivious dictionary data structure. Consider n documents identified with ids $\{d_1, d_2, \ldots, d_n\}$ and a set W of keywords extracted from these documents. For each $w \in W$, consider the sets of ids of documents which contain the keyword, denoted by $D_w = \{d_{w,1}, \ldots, d_{w,m_w}\}$, where $m_w = |D_w|$. The client employs a dictionary structure containing $(index, value)$ pairs to retrieve the set D_w for a given keyword w. This structure is built by by employing a PRF F and a symmetric cipher Γ, whose keys k_1 and k_2, respectively, are securely shared between the enclave and the client once they have established a secure channel. Specifically, for each keyword $w \in W$, m_w pairs are added to the dictionary, one for each document containing w:

$$\begin{cases} index_i = F_{k_1}(w \,||\, version \,||i) \\ value_i = \Pi.Enc(k_2, d_{w,i} \,||\, salt) \end{cases}$$

The client stores a local dictionary that associates each keyword w to a version and m_w, the number of documents containing $w \in W$,. When it needs to retrieve the ids of the documents containing w, it sends an encrypted token $(w, version, m_w)$ to the enclave, which can derive the $index_i$ for $i \in [1, m_w]$. Those indexes are used to fetch and remove $value_i$ from the dictionary, guaranteeing page fault and cache line obliviousness by relying on existing techniques to build an oblivious dictionary. The values are decrypted in the enclave using k_2 and transmitted to the client using the secure channel. The entries are re-added to the dictionary, with a different salt and an incremented version number (both on client and server). In order to update the dictionary with more entries, more $(index, value)$ pairs are added without updating the version number, but rather m_w. In this protocol, the enclave is just used as a safe storage for holding k_1 and k_2 and reduce the number of round trips of the protocol, similarly to embedding the ORAM client in an enclave. The information leakage assumed in Bunker is not concerned with the number of results, but guarantees that subsequent queries cannot be related: the indexes used to access the dictionary for a same keyword w change each time, and even if updates are performed without oblivious accesses, the protected fetch and remove procedure is sufficient to hide which dictionary entries match w. The crucial part of the protocol concerns updates: in particular, Bunker does not reveal the document ids of updates, but only their insertion time in the dictionary.

Chapter 3

Proposed Approach

We present Oblivious Substring Query on Remote Enclave (ObSQRE), a two-party protocol that allows a client to identify the positions of all the repetitions (or occurrences) of a substring in a text outsourced to a remote machine (e.g. in the cloud) in an oblivious manner. The obliviousness requirement mandates that the protocol guarantees the confidentiality of the outsourced text, the queried substring and the results of the query against an untrustworthy service provider, as well as protecting search pattern, which means that the could provider cannot correlate subsequent queries to establish their similarity. Our scenario, in which a memory constrained client owns a huge dataset over which she wants to perform several queries, suggests the usage of algorithms based on full-text indices, that allow to perform an arbitrary number of queries in sublinear time with respect to the size of the text and without applying any modification to the indexing structure itself. Since the construction of the index is performed offline, it doesn't impact the online efficiency of the protocol.

We structure the description of the protocol as follows: in section 3.1 we provide an overview of the opportunities and challenges posed by secure enclave, defining the rationale behind the implementation of *doubly oblivious* ORAMs; in section 3.2 we review all of our doubly oblivious constructions, showing their asymptotic complexity and optimizations enabled by our scenario; finally, in 3.3 we describe the full-text indices, how to pack them into ORAMs and the associated substring search algorithms.

3.1 ObSQRE Overview and Construction

A secure enclave is a trusted execution environment that runs within an untrusted system, giving the opportunity to have a TCB even on a remote server owned by a potentially malicious cloud provider. Since the memory of enclaves cannot be inspected by any hardware or software component on the machine, it is indeed possible to operate on sensitive plaintext data without any confidentiality concern. This feature totally cuts the necessity for complex cryptographic constructions such as FHE and SSE protocols, that instead perform computation directly on ciphertexts, with a performance overhead that usually prevents their adoption in real world scenarios. A program handling sensitive data which is run inside an enclave can fetch encrypted portions of the dataset and decrypt then in safe memory, thus allowing to combine outsourcing of data and the possibility to query it using already existing algorithms. Intel SGX is integrated on the vast majority of recent Intel CPUs, making enclaves widely available. Since they can run algorithms providing data confidentiality, the private subtring search problem may benefit from their exploitation as well. In particular, it is possible to outsource a huge text to the cloud, in encrypted form, while being able to find the occurrences of substrings. To provide high performance, it may be necessary to preprocess the text to generate an index that allows to run queries in sublinear time with respect to its length.

However, the rich literature on side channel attacks targeting Intel SGX proves the insufficiency of mere encryption in order to protect sensitive data, whose internal structure can be revealed by inspecting the way a client accesses her dataset even when these data are safely stored in enclave memory. In this regard, full-text indices require a lot of scattered memory accesses that may be analyzed by a malicious cloud provider to infer the sequence of logical memory identifiers, which depend on both the searched pattern and the full-text index, thus leaking much sensitive information. Hence, we need to devise specific countermeasures to protect confidential data as well as the computation that is performed on the server.

Rather than using compiler-based techniques or transactional memory to defeat side channel attacks, we embed sensitive information in ORAMs in order to conceal the logical access pattern to the indexing structures: this solution prevents an attacker from inferring both the content of the initial text and the similarities between subsequent queries. While ORAMs allow to decouple the logical identifiers of data structures from the memory access pattern, we also resort to algorithms whose control flow does not depend on secret information and that exhibit a data-independent execution trace. In other words, we design *oblivious* algorithms, which cannot be dismantled by the discovery of other side channels, and thus represents a long term solution. On the other hand, most of the side channel countermeasures presented in Section 2.1.2 are more exposed to potential vulnerabilities due to unknown architectural details, and address only one side channel at a time. The combination of several countermeasures might be inefficient, impossible or ineffective to face newer side channels.

Since ORAMs incur a very high bandwidth cost, we exploit enclaved execution to run within the server all the computation usually performed on the remote ORAM client. This cuts the necessity for network roundtrips in order to access the content of an ORAM, dramatically improving their performance. In particular, the number of matches *occ* can be obtained with a single communication. The actual indices of occurrences may be retrieved either with a single data exchange, or with a communication cost equal to $\mathcal{O}(occ)$. We implement the latter approach since it allows the client to fetch only a portion of the results, in case the specific match she is looking has already been found.

Nonetheless, embedding an ORAM client within an enclave requires relevant changes to the protocols since the shuffle operations are exposed to side channels as well. An adversary may come to know several information, such as whether a block is found in the stash or in the fetched path, the number of entries in the stash and, above all, the placement of data blocks inside buckets after evictions. This information, along with some domain knowledge about the usual access pattern of the application, allows a malicious server to find the mapping between leaf ids and block ids, thus subverting the security guarantees provided by ORAMs.

Traditional ORAMs constructions are usually referred to as *singly oblivious*: indeed, the fact that a block is reassigned a different path each time the client fetches it prevents the server from learning any information about the arrangement of the blocks in the ORAM: hence, she cannot correlate subsequent accesses and reconstruct the trace of logical indices used to fetch elements from data structures.

In order to make the protocol safe in our scenario, where the memory access pattern of the ORAM client is leaked ot the server, the client must be oblivious as well: hence, the resulting ORAM protocols are defined *doubly oblivious*. While doubly oblivious ORAM constructions already exist [24] [31], we extend this approach to Ring ORAM introducing optimizations that leverage the fact that the client resides on the same machine as the server. We implement doubly oblivious Path and Circuit ORAMs as well and review how to make them doubly oblivious.

Our constructions rely on two basic building blocks: an oblivious *ternary operator* and an oblivious Swap primitive. The former, `ternary_op`, allows to implement basic control flow without leaking which branch is chosen, while the latter, swap, is used to exchange the content of two data buffers based on a boolean condition without leaking the condition itself.

C++ ternary operator `cond?val_1:val_2` is an expression that evaluates to `val_1` if `cond` is true, to `val_2` otherwise. The prototype of its oblivious couterpart is `std::uint64_t ternary_op(bool sel, std::uint64_t a, std::uint64_t b)`. The parameters of the function follow the same ordering as the customary ternary operator. To introduce its implementation, we adopt the `System V AMD64 ABI` calling convention, as in all Unix-like systems, and the `AT&T` syntax for assembly. We exploit the `x86_64` instruction `CMOVcc` [31], that moves a source operand to a destination if the flag encoded in `cc` is set. Its property is that it always performs the same amount of work regardless of whether or not the destination operand is written, hence it is oblivious. Following the ABI, the three parameters of the function are stored in the registers `%rdi`, `%rsi`, `%rdx` respectively and the return value in `%rax`.

Listing 3.1: `ternary_op` function body

```
1 cmpb $0, %rdi
2 movq %rsi, %rax
3 cmovz %rdx, %rax
```

Listing 3.1 shows how the function is implemented. Line 1 evaluates the boolean condition: if `%rdi` is 0, `cmovz` overwrites the value in `%rax`.

Swap takes a boolean argument and two words made of k bits. Their values are swapped if the condition evaluates to true. Algorithm 3.1.1 shows its implementation. If *cond* is false, the value a and b are XORed with 0, and thus they do not change. On the other hand, if the condition is true, $a = a \oplus (a \oplus b) = b$, and similarly for b. The complexity of this algorithm is $\mathcal{O}(k)$.

Algorithm 3.1.1: Oblivious Swap primitive

```
Input: A boolean value $cond$
Input: A $k$-bit word $a$
Input: A $k$-bit word $b$
1 $mask \leftarrow$ ternary_op(cond,$1^k$, 0)
2 $temp \leftarrow mask$ & $(a \oplus b)$
3 $a \leftarrow a \oplus temp$
4 $b \leftarrow b \oplus temp$
```

The algorithm can be turned into an oblivious write primitive by suppressing line 4. In this case, the content of b, which becomes the source operand, overwrites a, which is the destination.

In order to obliviously access the position map, we either use an ODS or recursion. In the latter case, the chunks of position map that we fetch from the ORAMs are small enough to allow linear scans and oblivious selection of the proper leaf id for each level of the recursion. Since the size of the blocks of recursion is a free parameter, it is possible to tune this value to the one that balances the cost of linear scans and number of levels in the recursion. Stash management has a much higher impact on performance instead, because obliviousness makes evictions the most costly operation of doubly oblivious ORAMs. We will describe the details of our doubly oblivious contructions in the next Section 3.2.

Now that we have introduced doubly oblivious ORAM protocols, we need to clarify the terminology we will use to refer to the various actors of the system, since many ambiguities may arise. With the term *server*, we refer to the remote machine, possibly owned by a cloud provider, which is equipped with Intel SGX and runs the server-side of ObSQRE. The powerful threat model of Intel SGX includes root level adversaries that own the machine. Among the the various attacker models, a *honest but curious* adversary only monitors side channels to infer sensitive information, while a *malicious* one may deliberately tamper with the data or memory, deviating from the original protocol. In the latter case, the victim cannot prevent this behaviour, but may instrument the code in order to detect active attacks and abort the execution immediately. Performing a Denial of Service (DoS) attack would prevent the client from execute its program on remote server. If the server acts this way, it definitely loses the chance to monitor the queries performed by a client, and the opportunity to leak her secrets. Hence, DoS's are out of our scope.

The server executes the whole ORAM protocols locally. The *ORAM client* is embedded in an enclave, and accesses an ORAM tree that is kept in unprotected memory. The enclaved ORAM client fetches the correct paths itself from the unprotected memory: since data is not exchanged between physically separated machines, all the work is delegated to the enclaved portion of the protocol, even in the case of Ring ORAM. Ultimately, the *ORAM server* is just the unprotected memory that hosts the ORAM tree or an active attacker that deliberately tampers with it. On the other hand, the real *client* of the application is the entity who wants to perform substring search over a provided text. A totally separate machine hosts the client, who submits queries to the remote server through the network. The client is a very weak actor, with limited computational capabilities and memory.

Figure 3.1 shows the different interacting parts of ObSQRE. An Hypertext Transfer Protocol (HTTP) server exposes the functionalities of ObSQRE via API calls. The substring search algorithms run within the enclave and use the oblivious memory access primitives implemented in our library `libobl`. The indices for substring search are processed and encrypted offline, to prevent the server from inspecting them. The client uploads an index to the server by other means, prior to the execution of substring search protocol.

3.2 Design of Doubly Oblivious ORAMs

3.2.1 Doubly Oblivious Path ORAM

The Access procedure of Path ORAM is rather simple and requires little modifications in order to achieve obliviousness. The first difference is that an oblivious client cannot implement the stash as a dynamically sized linked list, since it would simply leak the number of elements that it

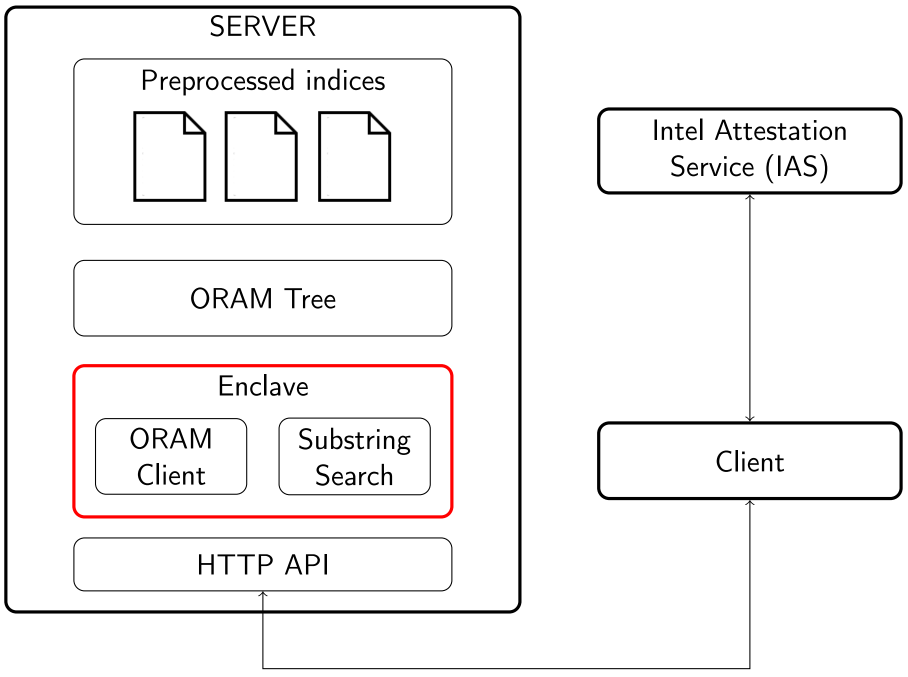

Figure 3.1: Overview of ObSQRE

contains during each access. On the other hand, in the original Path ORAM protocol, the stash has already a maximum size S established a priori to make overflow probability negligible. Therefore, in our doubly oblivious version, the stash has always a fixed size S, thus concealing the number of blocks stored. Furthermore, during both the fetch and eviction phase of the ACCESS routine, the stash is scanned linearly in order to conceal which element is actually retrieved as well as the position where a the block fetched from the path is placed. In this scenario, exceeding the stash size S becomes a severe failure condition, that also leaks the fact that the stash size grew up to $S + 1$ elements.

In the FETCH procedure, which aims at retrieving the block whose id is *bid*, the client linearly scans both the fetched path and the stash in order to find the proper record. In particular, to conceal which element is actually retrieved, the currently inspected block and the buffer which will eventually store the requested block are obliviously swapped, depending on the equality of the id of the block at hand with *bid*. The asymptotic cost of these operations is $\mathcal{O}(B \cdot (S + Z \cdot \log N))$.

During this procedure, the concatenation of the fetched path to the stash

(Algorithm 2.2.1, line 7) increases its size to $S' = S + Z \cdot \log N$. In this regard, we observe that the $Z \cdot \log N$ blocks are only temporarily stored in the stash: in fact, since they belong to a fetched path, they are already correctly bucketed because of the Path ORAM invariant. Hence, at least $Z \cdot \log N$ blocks will be evicted, keeping the stash size less than S after each ORAM access.

Algorithm 3.2.1 implements a baseline oblivious stash eviction. The client obliviously swaps each entry of the stash with every record of all the buckets, actually moving only the ones that can reside in the current bucket. To this extent, the function GetMaxDepth returns the maximum depth in the tree where a block can reside, given its leaf id and the current path. As already discussed in Subsection 2.2.1, the leaves are numbered in *reverse lexicographical order* and their id represents a path starting from the root. When two paths have common ancestors, they coincide up to a certain depth, where they diverge. By XORing together two leaf ids and counting the number of trailing zeroes, it is possible to calculate this depth. This operation requires $\mathcal{O}(1)$ on `x86_64` CPUs, that offer the an instruction that counts the trailing zeroes of a 64 bit word – namely `TZCNT`. Once again, to guarantee the best possible eviction probability, the blocks are pushed as down as possible in the path, that is filled starting from the leaf up to the root. The path that is evicted is initialized with dummy blocks, so that when a swap is successful, the corresponding entry of the stash becomes the dummy, and hence a free slot.

The complexity of this algorithm is:

$$\mathcal{O}\left(S' \cdot Z \cdot B \cdot \log N\right) = \mathcal{O}(S \cdot Z \cdot B \cdot \log N + Z^2 \cdot B \cdot \log^2 N\big)$$

A simple observation allows to easily improve the eviction complexity by a constant factor: in fact, all the blocks that belong to the path which has just been fetched are already correctly bucketed, since their leaf ids are not changed by the Access procedure. Therefore, only the block with id *bid* is obliviously (i.e., with a linear scan) removed from the path and appended to the stash, since it is reassigned a fresh leaf id that may otherwise break the path invariant. For all the other blocks in the fetched path, instead, we perform an *in place* eviction to push them as deep as possibile in the fetched

Algorithm 3.2.1: Oblivious Path ORAM OBLEVICT

```
   Input: The $lid$ of the path to evict
1  Initialize all the buckets in the path with block id $\bot$
2  for $i \leftarrow 0$ to $S' - 1$ do
3      $maxDepth \leftarrow$ GETMAXDEPTH($lid, stash[i].lid$)
4      for $l \leftarrow L - 1$ to $0$ do
5          $commonAncestor \leftarrow (maxDepth \geq l)$
6          for $z \leftarrow 0$ to $Z - 1$ do
7              $isFree \leftarrow (bucket[l].block[z].bid == \bot)$
8              $isValid \leftarrow (stash[i].bid \neq \bot)$
9              SWAP($isFree \wedge isValid \wedge$
               $commonAncestor, stash[i], bucket[l].block[z]$)
10         end
11     end
12     WRITEBUCKET($bucket, lid, l$)
13 end
```

paths: given a bucket at level i, we try to evict all of its blocks to all the buckets at levels $\{L-1, L-2, \dots, i+1]\}$, in order to place them in the deepest available record. The stash will no longer host S' slots, but only S (only the block retrieved by the client is appended to the stash), and its eviction is not changed at all.

The *in place* part of path eviction takes:

$$\mathcal{O}(\sum_{i=\log(N)-1}^{0} B \cdot Z^2 \cdot (\log(N)-1-i)) = \mathcal{O}(B \cdot Z^2 \cdot \sum_{i=0}^{\log N-1} i) = \mathcal{O}(B \cdot Z^2 \cdot \frac{\log^2 N}{2})$$

while the complexity of stash eviction is $\mathcal{O}(S \cdot B \cdot Z \cdot \log(N))$.

Oblix [24], which also employs a doubly oblivious Path ORAM, tries to further optimize the eviction procedure by hinging upon sorting networks, that are used to group the blocks in the stash by obliviously sorting them according to the max depth they can reach. At the end of the process, it is possible to rebuild a full path just by picking Z contiguous blocks at a time, which will form a new bucket. Of course, it is necessary to perform a preliminary analysis to include as many dummy blocks as needed to fill partially empty buckets. While this approach saves a factor of Z, the high constant factor hidden behind the asymptotic complexity of sorting networks

Algorithm 3.2.2: Oblivious Path ORAM OBLEVICT2

```
Input: An array of buckets $bucket$ after the fetched block has been
       replaced by a dummy
Input: The $lid$ of the path to evict
// in place path eviction
for $l_1 \leftarrow L-2$ to $0$ do
    for $z_1 \leftarrow 0$ to $Z-1$ do
        $maxDepth \leftarrow$ GETMAXDEPTH($lid, bucket[l_1].block[z_1].lid$)
        for $l_2 \leftarrow L-1$ to $l_1+1$ do
            $commonAncestor \leftarrow (maxDepth \geq l_2)$
            for $z_2 \leftarrow 0$ to $Z-1$ do
                $isFree \leftarrow (bucket[l_2].block[z_2].bid == \perp)$
                $isValid \leftarrow (bucket[l_1].block[z_1].bid \neq \perp)$
                SWAP($isFree \wedge isValid \wedge$
                $commonAncestor, bucket[l_2].block[z_2], bucket[l_1].block[z_1]$)

            end
        end
    end
end
for $i \leftarrow 0$ to $S-1$ do
    $maxDepth \leftarrow$ GETMAXDEPTH($lid, stash[i].lid$)
    for $l \leftarrow L-1$ to $0$ do
        $commonAncestor \leftarrow (maxDepth \geq l)$
        for $z \leftarrow 0$ to $Z-1$ do
            $isFree \leftarrow (bucket[l].block[z].bid == \perp)$
            $isValid \leftarrow (stash[i].bid \neq \perp)$
            SWAP($isFree \wedge isValid \wedge$
            $commonAncestor, stash[i], bucket[l].block[z]$)
        end
    end
    WRITEBUCKET($bucket, lid, l$)
end
```

(which is $\mathcal{O}(S \log^2 S)$), makes this solution slower than the naive approach.

Following this idea, we also tried to further optimize eviction: in particular, we noticed that we can perform evictions without linear scans over both the stash and the evicted path if all of their elements are obliviously shuffled. Indeed, after the shuffle, an attacker cannot distinguish real and dummy blocks, as well as the bucket originally storing them. We devised an oblivious shuffle algorithm, outlined in Algorithm 3.2.3, that operates on arrays whose size is a power of 2, a requirement that can be easily fulfilled via padding. The algorithm divides the array in two partitions and obliviously swaps the corresponding elements in each partition. At the end of this process, each element has exactly the same probability of residing in either of the two portions of the array. Then, the same algorithm is recursively applied to the two subarrays, until only one element is left in each partition. The movements of each entry between partitions is akin to the exploration of a complete binary tree from root to leaf, where the left or right path is taken with equal probability of 1/2. Hence, at the end of the process, each leaf (i.e. the final position in the shuffled array) has probability $1/N$ of being reached. When N is not a power of two, the resulting recursion tree is unbalanced and thus the algorithm does not provide uniform probability. Although

Algorithm 3.2.3: Oblivious Shuffle

```
   Input: An array $A$ whose size is a power of 2
1  if $|A| == 1$ then
2      return
3  end
4  else
5      for $i \leftarrow 0$ to $\frac{|A|}{2} - 1$ do
6          $cond \leftarrow RandomBit(0, 1)$
7          $j \leftarrow \frac{|A|}{2} + i$
8          Swap($cond == 0, A[i], A[j]$)
9      end
10     Shuffle($A[0 \dots \frac{|A|}{2} - 1]$)
11     Shuffle($A[\frac{|A|}{2} \dots |A| - 1]$)
12 end
```

the complexity of Algorithm 3.2.3 for an array of size n is $\mathcal{O}(n \log n)$ swaps, the performance we achieved are much worse than the baseline, confirming the experimental results of [24]. We thus opted for the improved version of eviction as implemented in Algorithm 3.2.2.

3.2.2 Doubly Oblivious Ring ORAM

The Ring ORAM protocol is elaborate with respect to Path ORAM, and manages to reduce the online bandwidth to $\mathcal{O}(1)$ thanks to complex mechanisms that rely on metadata associated to each bucket, as reported in Table 2.1. Ring ORAM performs an eviction each A accesses and in order to avoid stash growth, endows each bucket with D dummy blocks that are fetched when the one with the correct block id is not found. The D dummy blocks and Z possibly valid records are intermixed in order to prevent the remote server from understanding whether or not a bucket contains the requested block. In order to keep track of the position of the non-dummy blocks, their offsets in the buckets are stored in the PRP field. Since after D accesses a bucket potentially runs out of dummies, the EarlyReshuffle procedure refreshes them in order to prevent more that one valid block to be fetched during each invocation of Access.

Even though the doubly oblivious implementation of Ring ORAM we provide abides by this framework, some modifications are necessary to improve it based on the fact that the ORAM client resides on the same machine as the server. The most relevant difference between the original version of Ring ORAM and our implementation is that the buckets don't store the PRP of the Z valid blocks. In fact, if keeping track of the offsets where real blocks reside is not strictly necessary for a *singly oblivious* implementation, it is totally unnecessary when the client is hosted within an enclave. In the original SelectOffset procedure of Ring ORAM, the PRP is employed to distinguish the Z real blocks from the D dummy ones. Specifically, the PRP allows to look for dummy blocks only by iterating over D validity bits (Algorithm 2.2.5, line 4) and, similarly, to verify if a real block is valid by directly looking at its valid bit (Algorithm 2.2.5, line 7). Nonetheless, in a doubly oblivious scenario, the algorithm can no longer access only the D validity bits of the dummy blocks, as this would trivially leak the positions of the real blocks, making the PRP pointless. Therefore, we discard the usage of the PRP, resorting to the analysis of the block ids in order to distinguish between a real block and a dummy block. Specifically, the bucket metadata stores the block id of all blocks, including the dummy ones, slightly increasing the size of the metadata: indeed, our doubly oblivious version saves the $Z \cdot \log(Z + D)$ bits required by the PRP but stores D additional block ids, which require $D \cdot \log(N)$ space. However, this equivalent way to encode bucket metadata allows to select the proper offset for a bucket with a single sweep of the bucket metadata, as showed in Algorithm 3.2.4. Our SELECTOFFSET procedure picks a specific record from each bucket, either choosing a dummy block or the correct one, if valid. The flag $found$ returned by the procedure specifies if the selected block is a real or a dummy one. The procedure needs to ensure that index selection looks random: while real records are scattered for this purpose, the client must choose a totally random dummy, possibly with a single scan over the block ids. In order to achieve this purpose, the client extracts a random number, and selects the current dummy record only if this value is greater than the previous maximum. Of course, if the requested block id is found, the client chooses that index instead. Since the selection of a dummy index depends on a random value, it looks random as well.

Whenever the client finds the $(offset, found)$ pair for a bucket, it fetches the block identified by $offset$, and obliviously writes it to a buffer only if $found$ evaluates to true. This cuts the necessity for the XOR trick, since the block stored in the buffer after all buckets of the path have been analyzed is the only one actually written, which corresponds to the only real block selected among the buckets in the path. Finally, since dummy blocks don't require to be the encryption of a known string, they can be left uninitialized. We remark that this approach blurs the distinction between dummy blocks inside the Z set and the dummies of D: however, if during eviction we guarantee that a bucket has at most Z valid blocks, our doubly oblivious construction is semantically equivalent to a baseline Ring ORAM.

Concerning the eviction of a path, we recall that in Ring ORAM the client only needs to evict Z blocks for each bucket, since this is the number of possibly non-dummy blocks. It may happen that there are more than Z valid records when the bucket being evicted has been accessed less than D times: in this case, some of the remaining records will be discarded in order to leave exactly Z valid blocks. In order to do that, the client samples at random the blocks to invalidate, choosing among the valid ones (i.e. that

Algorithm 3.2.4: Oblivious SelectOffset function

```
Input: A block id $bid$
Input: Encrypted metadata $meta$ of a bucket
Output: The chosen offset in the bucket and whether it is a dummy
1  $dec\_meta \leftarrow \textsc{Decrypt}(meta)$
2  $dec\_meta.counter \leftarrow dec\_meta.counter + 1$
3  $curr\_max \leftarrow -1$
4  $found \leftarrow$ false
5  $offset \leftarrow \textsc{UniformRandom}(0, Z + D - 1)$
6  for $i \leftarrow 0$ to $Z + D - 1$ do
       // the server knows which blocks were already accessed
7      if $dec\_meta.valid[i]$ then
           // the range [0,127] is a mere implementation choice
8          $rnd \leftarrow \textsc{UniformRandom}(0, 127)$
           /* always evaluates to false once the bid has been
              found in the bucket                              */
9          $upd\_dummy \leftarrow (dec\_meta.bid[i] == \perp) \wedge (rnd > curr\_max) \wedge \neg found$
10         $upd\_found \leftarrow (dec\_meta.bid[i] == bid)$
11         $offset \leftarrow$ ternary_op(upd_dummy, i, offset)
12         $curr\_max \leftarrow$ ternary_op(upd_dummy, rnd, curr_max)
13         $offset \leftarrow$ ternary_op(upd_found, i, offset)
14         $found \leftarrow found \vee upd\_found$
15     end
16 end
17 return (offset, found)
```

haven't been accessed yet) with block id $\perp$. Knuth's S algorithm [23, pag. 142] samples with uniform probability a required number of elements out of a set in a single scan. Because of these properties, its oblivious implementation is straightforward. Once that the Z records are read into the enclave, the same eviction procedure as the Path ORAM applies: the ORAM client first performs an in place eviction, and then a stash eviction, which tries to place as many entries as possible from the stash in the evicted path. The only difference is that each block is obliviously written to only one uniformly randomly chosen location per bucket, saving a multiplicative factor of Z. This is possible since the Z blocks will be later permuted and interspersed in $Z + D$ records, in order to reconstruct the original bucket structure. Since this is akin to a reshuffle, it is not necessary to conceal the block offset where a record is evicted.

Before writing back a path to the ORAM tree, the client needs to permute the Z possibly valid entries in each bucket of the path. In our case, the client intersperses them into $Z + D$ slots, guaranteeing that each valid block will be written to a location with uniform probability. At the beginning, the $Z + D$ entries of the bucket are empty, i.e. their block id is $\perp$. We perform Z iterations in which each block is obliviously swapped with every block of the bucket. Only one of these writes will be real. In order to ensure that each of the real block is written to a specific location of the bucket with uniform probability, we resort to Algorithm 3.2.5. During its i-th iteration, there will be $Z + D - i$ dummy blocks in the bucket. The client draws a random integer $offset \in [0, Z + D - i - 1]$ and writes the valid block to the $offset$-th unselected block in the bucket. In order to do that, the client subtracts 1 from $offset$ each time an unwritten block is met. When $offset$ drops to -1, a real write is carried out since the correct entry has been reached. The next times an unselected block is found, the value of $offset$ will decrease further, thus not triggering the write condition in line 10.

It is possible to prove that the presented algorithm guarantees uniform distribution of the Z real blocks in the bucket. The j-th record in the bucket will host the i-th record if the client extracts it at the i-th iteration. The procedure is equivalent to drawing, at each stage, one random offset x among the unselected ones. If we assume that the generated random values are not

Algorithm 3.2.5: Oblivious INTERSPERSE function

```
Input: An array of $Z$ blocks $block$
Output: An array of $Z + D$ blocks $oblock$
1  for $i \leftarrow 0$ to $Z + D - 1$ do
2      $oblock.bid[i] \leftarrow \perp$
3      $selected[i] \leftarrow$ false
4  end
5  for $i \leftarrow 0$ to $Z - 1$ do
6      $offset \leftarrow$ UNIFORMRANDOM$(0, Z + D - i - 1)$
7      for $j \leftarrow 0$ to $Z + D - 1$ do
           // a block not selected is a dummy for sure
8          $isDummy \leftarrow \neg selected[j]$
9          $offset \leftarrow offset -$ ternary_op(isDummy, 1, 0)
10         $pick \leftarrow offset == -1 \wedge isDummy$
11         $selected[j] \leftarrow selected[j] \vee pick$
12         SWAP$(pick, oblock[j], block[i])$
13     end
14 end
15 return $oblock$
```

correlated, this probability is:

$$Pr[x_i = j \wedge x_{i-1} \neq j \wedge \cdots \wedge x_0 \neq j] = \frac{1}{Z+D-i} \cdot \prod_{k=0}^{i-1} \frac{Z+D-k-1}{Z+D-k} = \frac{1}{Z+D}$$

The complexity of Intersperse is $\mathcal{O}(B \cdot Z \cdot (Z+D))$. The overall complexity of evictions, considering that it is performed every A accesses, is

$$\mathcal{O}\left(\frac{B \cdot Z \cdot \log^2 N}{2 \cdot A} + \frac{B \cdot S \cdot \log N + B \cdot Z \cdot (Z+D) \cdot \log N}{A}\right)$$

In the original Ring ORAM paper [16], the authors propose to append the Z valid blocks remaining in the bucket to the stash and then tries to evict Z blocks from the stash in this bucket, in order to reduce the stash occupancy. However, the stash overflow analysis of Ring ORAM [16] assumes that reshuffles don't influence stash occupancy, hence the implementation they provide for sure reduce stash occupancy, but is not required to prevent stash overflows. Since in our scenario evicting Z blocks from the stash would require Z costly linear scans of the stash, our EarlyReshuffle just calls Intersperse for all the buckets in the path that have been accessed more than D times. We empirically verified the statements reported in the stash overflow analysis, and never incurred an overflow due to this implementation.

3.2.3 Doubly Oblivious Circuit ORAM

Circuit ORAM exhibits a very high bandwidth blow-up since it needs to fetch three paths for every access. The first one contains the block that the client wants to retrieve, while the other two paths are selected by the deterministic eviction schedule. This is a major issue that makes of Circuit ORAM impractical for remote protocols, but since bandwidth is not a problem when the client and the server reside on the same physical machine, be it on a circuit or within an enclave, it well suits our scenario.

In order to prevent side-channel leakage when the adversary has physical access to the chip implementing the client logic, Circuit ORAM routines, and in particular evictions, are designed to be doubly oblivious by default. In particular, Algorithms 2.2.6 and 2.2.7 are oblivious when the assignments enclosed in `if` statements are substituted by our oblivious ternary operator.

Since they perform a linear scan over the stash and fetched path, without moving any data block, their complexity is $\mathcal{O}(S + Z \cdot \log N)$. The whole eviction, outlined in Algorithm 2.2.8, requires little modifications, which are showed in Algorithm 3.2.6: in particular, the client obliviously swaps the block contained in *to_write* with every record of the current bucket, in order to conceal the target bucket of the block currently stored in *hold*. This sequence of operations is marked in red in the pseudocode and is performed for every block of a bucket, yielding an additional factor of Z in the asymptotic complexity. Hence, the resulting complexity we achieve is $\mathcal{O}(B \cdot (S + Z \cdot \log N))$, which prevails on the two linear scans over metadata. Even though this is the best eviction complexity achieved, it is necessary to take into account that the client applies this procedure twice, hence its efficiency is not visible for small ORAMs, but rather asymptotically.

One great advantage is that the stash size is very low with respect to the other protocols, making linear scans much faster.

3.2.4 Protection against active attackers

The ORAMs presented so far guarantee confidentiality and protect the user from a *honest but curious* attacker, but don't provide any guarantee against intentional memory tampering. Authenticated Encryption (AE) provides integrity and authentication, thus allowing to detect when the ciphertext contained in the ORAM tree has been altered. However, it is insufficient in preventing a malicious attacker from delivering more refined techniques, such altering the topology of the tree by shuffling buckets, or replaying older values. These attacks would remain undetected since the MAC stored inside bucket metadata would match, as AE alone cannot ensure *freshness* of encrypted data, nor fix the structure of the tree. To achieve this purpose, the ORAM tree can be enriched to implement a Merkle tree with data [27].

Given a hash function H, a Merkle tree with data is a binary tree which stores within each internal node a digest $H_n = H(data \,||\, H_l \,||\, H_r)$, where H_l and H_r are the digests contained in the left and right child, respectively. As an ORAM is organized as a binary tree too, each bucket of the ORAM can be the *data* stored in a Merkle tree node. This construction allows the client to verify the integrity of all the paths in the ORAM tree by retaining only the digest of the root node. Indeed, this digest can be re-computed from

Algorithm 3.2.6: Circuit ORAM Eviction

```
   Input: An array of buckets $bucket$
   Input: The leaf id $lid$ of the current path
   Input: The output of Algorithm 2.2.7 $target$
1  $hold \leftarrow \perp$
2  $dst \leftarrow target[-1]$
3  $max\_depth \leftarrow -1$
4  for $i \leftarrow 0$ to $S-1$ do
5      $curr\_depth \leftarrow$ GetMaxDepth($stash[i].lid, lid$)
6      $deepestBlock \leftarrow (curr\_depth > max\_depth) \wedge (stash[i].bid \neq \perp) \wedge target[-1] \neq \perp$
7      $max\_depth \leftarrow$ ternary_op(deepestBlock, curr_depth, max_depth)
8      Swap($deepestBlock, hold, stash[i]$)
9  end
10 for $l \leftarrow 0$ to $L-1$ do
11     $toEvict \leftarrow (l == dst) \wedge (hold \neq \perp)$
12     $dst \leftarrow$ ternary_op(toEvict, target[l], dst)
       // if to evict, immediately move to bucket
13     Swap($toEvict, hold, bucket[l].block[0]$)
14     $curr\_depth \leftarrow$ GetMaxDepth($bucket[l].block[0].lid, lid$)
15     $cond \leftarrow toEvict \wedge bucket[l].block[0].bid \neq \perp$
16     $max\_depth \leftarrow$ ternary_op(cond, curr_depth, -1)
17     for $i \leftarrow 1$ to $Z-1$ do
18         $curr\_depth \leftarrow$ GetMaxDepth($bucket[l].block[i].lid, lid$)
           // if the next dst is $\perp$
19         $swap\_cond \leftarrow (dst == \perp) \wedge (bucket[l].block[i].bid == \perp) \wedge (hold.bid \neq \perp)$
           // if the next dst is $\neq \perp$
20         $swap\_cond2 \leftarrow (dst \neq \perp) \wedge (bucket[l].block[i].bid \neq \perp) \wedge (curr\_depth > max\_depth)$
21         Swap($toEvict \wedge (swap\_cond \vee swap\_cond2), bucket[l].block[i], hold$)
22         $max\_depth \leftarrow$ ternary_op(swap_cond2, curr_depth, max_depth)
23     end
24 end
```

all the nodes of each path, starting from the leaf up to the root. Therefore, given a fetched path from the ORAM, the integrity of all nodes on the path can be verified by re-computing the digest of the root node and comparing it with the digest retained by the client. Thanks to its structure, a Merkle tree allows the client to verify any tampering of the ORAM tree topology and any bucket swap results in a failure of the digest verification. The digest computation has no additional overhead when using an AE scheme, such as AES in Galois Counter Mode (GCM). Indeed, these schemes produce also a cryptographic digest which is verified upon decryption of the data.

The disadvantage of this construction is that, in order to compute the digest for all the nodes, all the buckets in the ORAM (even dummy ones) need to be fill with some data (i.e., initialized), an operation that may become slow as its size grows (although it would be performed just once when constructing the ORAM tree). Luckily, it is possible to circumvent this drawback [27] by adding two boolean values, v_l and v_r to each internal node. These values tell whether or not the left or right child, respectively, has been accessed at least once. The buckets that have never been accessed are left uninitialized, and client will fill them with actual content only the first time a path containing these buckets is evicted. In case an uninitialized bucket is found along a path, its digest is substituted by 0. The tag associated to each node is thus

$$H_n = H(data \,||\, v_l \,||\, v_r \,||\, v_l \wedge H_l \,||\, v_r \wedge H_r)$$

The server knows which buckets were already accessed by the client, since they reside in unprotected memory: as a consequence, it also knows which ones are uninitialized. Even though the digest of invalid buckets is not checked, the server cannot exploit this condition to inject its own data, as the client totally discards their content.

The Circuit and Path ORAM that we implement embed this mechanism to ensure freshness and protection against a malicious attacker. Ring ORAM, instead, forces the usage of CTR Mode mode to enable scattered access to data blocks and doesn't decrypt a whole bucket when a single record is fetched. These characteristics make it unsuitable for the integration of a Merkle tree: as a consequence, it only guarantees protection against a honest

	$	**a**	**f**	**l**
a	0	0	0	0
f	0	1	0	0
f	0	1	1	0
$	0	1	2	0
l	1	1	2	0
l	1	1	2	1
a	1	1	2	2
a	1	2	3	2
	1	3	3	2

Figure 3.2: BWT rank matrix for `alfalfa`

but curious adversary.

3.3 Substring search algorithms

The algorithms of substring search we implement are variations of backwards search (Algorithm 2.3.5). Its advantage is that it is a relatively simple algorithm that requires much more compact data structures than suffix trees, but yields the same time complexity. Moreover, its control flow is extremely regular and can be made oblivious just by exploiting ORAMs to access indexing structures.

3.3.1 BWT based substring search

The first algorithm we present relies on the BWT of the original text T. It performs rank queries in $\mathcal{O}(1)$ thanks to precalculated values stored in a matrix, that will be referred to as M. M has $|T|+2$ rows and $|\Sigma|+1$ columns. Each entry $M[x][\sigma]$ stores $rank_{BWT,\sigma}(x)$, thus reducing rank queries to a simple lookup. Figure 3.2 shows the resulting indexing data structure for our running example `alfalfa`. We remark that the last row is needed since $rank_{BWT,\sigma}(x)$ counts the occurrences of σ in $BTW[0, x-1]$, thus without

$count_0$	0	0	0	0
BWT_0	**a**	**f**	**f**	
$count_1$	0	1	2	0
BWT_1	**\$**	**l**	**l**	
$count_2$	1	1	2	2
BWT_2	**a**	**a**	ϵ	

Figure 3.3: BWT rank matrix for `alfalfa` sampled with period $R = 3$

the last row, the occurrence of the last character in the BWT will not be counted in any of the entries of the matrix M.

The size of M may become prohibitive for large texts and alphabets, since it contains $\mathcal{O}(|\Sigma| \cdot |T|)$ entries whose size in bits is $\mathcal{O}(\log |T|)$, as a single character σ may occur up to $|T|$ times in the original text. To overcome this issue, we construct a new data structure which retains only some of the rows of M, interleaving these rows with the BWT of the text. The construction of this data structure depends on an integer R, referred to as *sample period*, which determines which rows of M are kept. Specifically, the data structure M_R has $\lceil \frac{|T|+2}{R} \rceil$ entries, where the x-th entry is a pair of elements $(count_x, BWT_x)$, the former being the $(x \cdot R)$-th row of M and the latter being a substring of the BWT from position $x{\cdot}R$ to position $(x{+}1){\cdot}R{-}1$ (i.e., $count_x = M[x \cdot R,]$ and $BWT_x = BWT[x \cdot R, x \cdot R + R - 1]$). Figure 3.3 shows the matrix M_R for sampling period $R = 3$. In the last sample, BWT_3 is padded with the empty character ϵ to exactly count 3 characters.

We now describe how to compute the $rank_{BWT,\sigma}(x)$ by hinging upon M_R. Let idx be $\lfloor x/R \rfloor$. In order to perform the rank query, it is sufficient to fetch the idx-th entry of M_R and compute the sum of $count_{idx}[\sigma] = M[idx \cdot R, \sigma]$ with the number of occurrences of σ in $BWT_{idx} = BWT[idx \cdot R, idx \cdot R + R - 1]$. Algorithm 3.3.1 shows the resulting rank procedure. Even if it scans up to R characters of the BWT, the size of R is usually negligible with respect to $|T|$, so that its complexity is still $\mathcal{O}(1)$. We observe that M_R improves the required space only by the constant factor R, implying that the asymptotic bounds for memory are the same. However, since linear scans are very fast, R can be chosen big enough to make a huge difference in practical implementations. The tradeoff between computation and space saving can be tuned freely according to the requirements of the application.

Algorithm 3.3.1: RANK using M_R

```
Input: The index $x$ of the rank
Input: The character $\sigma$ of the rank
Input: The BWT of $T$
Input: The matrix $M_R$
Output: $rank_{BWT,\sigma}(x)$
1  $idx \leftarrow \lfloor x/R \rfloor$
2  $offset \leftarrow x \mod R$
3  $partial\_rank \leftarrow 0$
4  for $i \leftarrow 0$ to $offset-1$ do
5      if $M_R[idx].BWT[i] = \sigma$ then
6          $partial\_rank \leftarrow partial\_rank + 1$
7      end
8  end
9  $rank \leftarrow M_R[idx].count[\sigma] + partial\_rank$
10 return $rank$
```

While R is a public parameter of our protocol. when Algorithm 3.3.1 is executed inside an enclave, we need to conceal which entry of M_R is accessed, as well as the σ entry of $count_{idx}$ and the value of $offset$. To conceal idx, we wrap the data structure M_R into a recursive ORAM, storing both $count_{idx}$ and BWT_{idx} in a single ORAM block. Furthermore, to conceal the entry accessed in $count_{idx}$, we resort to a linear scan that selects the element corresponding to character σ with oblivious swaps. Finally, we always perform R iterations in Algorithm 3.3.1 to hide the value of $offset$, using the `ternary_op` primitive to inhibit the increment when $offset$ characters of the BWT_{idx} have already been scanned. Backwards search also needs the $\mathcal{C}$ array: since its length is $|\Sigma| + 1$, it is small enough to allow linear scans to hide which entry is actually fetched.

The access to the recursive ORAM impacts the performance of this algorithm. In a doubly oblivious scheme, the ACCESS cost is dominated by the eviction procedure: hence, in order to estimate the complexity of our oblivious algorithm, we choose to employ the eviction complexity of Circuit

ORAM, as it is the most efficient. From this complexity, we omit the terms S, the stash size, and Z, the number of blocks found in each bucket, as they are always small enough to be considered negligible with respect to the size of the ORAM. Therefore, we employ $\mathcal{O}(B \cdot \log N)$ as the cost of an ORAM access. In Subsection 2.2.4 we evaluated the cost to retrieve the leaf id from a recursive position map, which is $\mathcal{O}(B_R \cdot \log_2^2 N / \log_2 C)$, where B_R is the block of the recursive ORAMs employed to store the position map and C is the recursion factor for the recursive ORAMs. This bound was assessed for a remote ORAM protocol, whose bandwidth cost is the figure of merit of its complexity and amounts to $\mathcal{O}(B \cdot \log N)$. The bounds we found for the remote scenario also hold for Circuit ORAM, even though its asymptotic complexity is related to the cost of evictions. Since doubly oblivious Path and Ring ORAM have complexity that is roughly $\mathcal{O}(B \cdot \log^2 N)$, recursive access to the position map has a cost of $\mathcal{O}(B_R \cdot \log_2^3 N / \log_2 C)$. The number of elements of the last level ORAM is $|T|/R$, the size of the block of the recursive position map is $B_R \in \mathcal{O}(C)$, while the block size of the ORAM containing M_R is $B_0 \in \mathcal{O}(|\Sigma| + R)$.

In conclusion, the cost to access the ORAM wrapping the M_R data structure is:

$$\mathcal{O}\left(\frac{C}{\log_2 C} \cdot log_2^2\left(\frac{|T|}{R}\right) + (|\Sigma| + R) \cdot \log_2\left(\frac{|T|}{R}\right)\right) = \mathcal{O}\left(log_2^2(|T|)\right)$$

when we consider C, R and Σ constants. When searching a pattern of size m, we obtain $\mathcal{O}(m \cdot \log^2(|T|/R))$.

3.3.2 STBWT algorithm

An alternative way of performing rank queries would be storing, for each character $\sigma \in \Sigma$, an array occ_σ whose entries are the positions where a it occurs in the BWT, in order. If we have such information, the $rank_{BWT,\sigma}(x)$ would trivially be the first index k such that $occ_\sigma[k] \geq x$. The various occ_σ arrays drawn from the BWT of `alfalfa`, whose $\Sigma = \{a, f, l\}$, are represented in Figure 3.4.

Since occ_σ is sorted, we can improve on this approach by performing a rightmost binary search for x; the outcome of this search will be the last index h such that $occ_\sigma[h] < x$, therefore the resulting rank will be $k =$

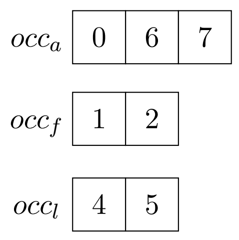

Figure 3.4: Array of the occurrences for the BWT of `alfalfa`

$h+1$, as the first index of the array is 0 and we actually want to count the number of entries before the index k. The binary search procedure is reported in Algorithm 3.3.2, which can just provide *rank* calculations in Algorithm 2.3.5. We refer to the resulting algorithm as Search-tree BWT (STBWT). Since the cardinality of the different occ_σ arrays is proportional to the length of the text, the complexity of the unprotected version of this algorithm is $\mathcal{O}(m \cdot \log N)$.

Algorithm 3.3.2: STBWT RANK

```
Input: The array of the occurrences $occ_\sigma$
Input: The argument $x$ of the rank
Output: $rank_{BWT,\sigma}(x)$
$st \leftarrow 0$
$end \leftarrow |occ_\sigma| - 1$
$rank \leftarrow 0$
while $st <= end$ do
    $middle \leftarrow st + \lfloor \frac{end-st}{2} \rfloor$
    if $occ_\sigma[middle] < x$ then
        $st \leftarrow middle + 1$
        if $middle \geq rank$ then
            // indexing starts from 0, so we add 1
            $rank \leftarrow middle + 1$
        end
    end
    else
        $end \leftarrow middle - 1$
    end
end
return $rank$
```

In order to hide the access pattern to the array, it is necessary to wrap it within an ORAM. A convenient way to achieve this purpose is exploiting the general framework of tree-based ODS's. In fact, the binary search performed over the occ_σ arrays is akin to the exploration of a search tree: building a search tree from a sorted array is a trivial task, that only requires to recreate the access pattern of the binary search. In particular, for each partition delimited by the indices *st* and *end*, that defines the nodes contained in a subtree, the middle element evaluated as the entry at index $middle = st + \lfloor \frac{end-st}{2} \rfloor$ is chosen as subroot, splitting the given subtree in two other partitions delimited by $st, middle - 1$ and $middle + 1, end$ respectively. Figure 3.6(b) shows the binary tree generated from the sorted array in (a). The numbers shaded in black represent the indices of the initial array that hold the search key, that are equivalent to the value of *middle* for their corresponding partition. Of course, this approach yields a balanced search tree whose depth is $\log |occ_\sigma|$, which in fact derives from the time complexity of a binary search over a sorted array. For each $\sigma \in \Sigma$ it is thus necessary to build a separate tree: however, it is not possible to wrap them within distinct ODS's, since the current pattern character would be trivially leaked by detecting which ODS is currently accessed by the enclave. Hence, in the case of STBWT, all the search trees are kept in a single ODS: in order to explore each one of them, we keep an array of $|\Sigma|$ elements that point to the various roots. Once that the ORAM address of a root has been resolved, the exploration continues as in a standard ODS, leveraging the pointers contained within each internal node, which references the children elements of the node at hand in a single tree. In order to obliviously select the proper root pointer, we resort to linear scans: this is possible since the number of roots $|\Sigma|$ is usually negligible with respect to the length of the text.

The details of our ODS construction are deferred to Subsection 3.3.4, as well as the resulting complexity of a root to leaf exploration performed by STBWT.

3.3.3 SA-Ψ algorithm

The successor array Ψ is a data structure that derives from the suffix array and allows to implement an alternative version of backwards search. If $SA[x]$ is the suffix of T that starts at the k-th character of the string, $\Psi[x]$

a)

7	6	3	0	5	2	4	1

b)

3	7	5	2	6	4	1	0

c)

$\perp$	0	6	7	1	2	4	5

Figure 3.5: (a) The SA (b) ISA and (c) Ψ of the running example `alfalfa`

is the index of SA of the suffix starting at position $k + 1$:

$$SA[\Psi[x]] = SA[x] + 1$$

The construction of the Ψ array is quite straightforward. Let ISA be the inverse suffix array: if $SA[x] = k$, then $ISA[k] = x$. In other words, the k-th entry of the ISA tells which index of the SA stores the suffix starting at the k-th character of T. The construction of the ISA is shown in Algorithm 3.3.3. It turns out that $\Psi[x]$ is, in fact, $ISA[SA[x] + 1]$, save for $\Psi[0] = \perp$. In fact, $SA[0]$ corresponds to the terminator \$, which has no successors. The ISA and the successor array Ψ for string *alfalfa* are depicted in Figure 3.5.

Algorithm 3.3.3: Construction of the ISA

```
Input: The suffix array $SA$
Output: The inverse suffix array $ISA$
1 for $i \leftarrow 0$ to $|SA| - 1$ do
2     $ISA[SA[i]] \leftarrow i$
3 end
4 return $ISA$
```

Given a character $\sigma \in \Sigma$, the entries of Ψ in the range $[\mathcal{C}[\sigma], \mathcal{C}[next(\sigma)]-1]$ point to the successors of all the suffixes that start with σ. If we consider two consecutive entries of the suffix array in that range, namely $SA[x] =$

$k_1, SA[x{+}1] = k_2$, they point to two suffixes that can be written as $suffix_{k_1} = \sigma \,||\, suffix_{k_1+1}$ and $suffix_{k_2} = \sigma \,||\, suffix_{k_2+1}$. Of course, $\Psi[SA[x]]$ will point to $suffix_{k_1+1}$ while $\Psi[SA[x+1]]$ to $suffix_{k_2+1}$. Since $suffix_{k_1}$ is lexicographically smaller than $suffix_{k_2}$ (being the suffix array sorted), the suffixes obtained by dropping their initial character σ will preserve this ordering as well: hence, the suffix pointed by $\Psi[SA[x]]$ is smaller that the one of $\Psi[SA[x+1]]$. Repeating this reasoning for all the contiguous entries of the Ψ array that fall in the range $[\mathcal{C}[\sigma], \mathcal{C}[next(\sigma)] - 1]$, we obtain a key property which will be exploited in the search algorithm, namely that the Ψ entries relative to the same character are sorted, i.e., for each $\sigma \in \Sigma$, $\Psi[\mathcal{C}[\sigma], \mathcal{C}[next(\sigma)] - 1]$ is sorted in ascending order.

The concept behind backwards search with Ψ relies on this property. As usual, given a pattern P of length m, the search starts from its last character, including a new one at each iteration. In the i-th iteration, the backwards search algorithm generates the indices s_{m-i} and e_{m-i} that delimit the portion of the suffix array which contains the positions of suffixes whose prefix is the pattern $P[i, m-1]$. The boundaries for the last character $P[m-1] = \sigma_{m-1}$ are retrieved from the $\mathcal{C}$ array as usual: $s_{m-1} = \mathcal{C}[\sigma_{m-1}]$ and $e_{m-1} = \mathcal{C}[next(\sigma_{m-1})] - 1$. If we include the next character $P[m-2] = \sigma_{m-2}$, the refined limits will fall in the range $\{\mathcal{C}[\sigma_{m-2}], \mathcal{C}[next(\sigma_{m-2})] - 1\}$, which indeed points to all the suffixes of the text starting with σ_{m-2}. Among them, we want to include only the ones whose second character is σ_{m-1}. This requirement is fulfilled by all the suffixes whose Ψ entry falls in the range $\{s_{m-1}, e_{m-1}\}$. Since $\Psi[\mathcal{C}[\sigma_{m-2}], \mathcal{C}[next(\sigma_{m-2})] - 1]$ is sorted, then these suffixes are represented by two indexes s_{m-2}, e_{m-2}: the former is the position of the leftmost element in $\Psi[\mathcal{C}[\sigma_{m-2}], \mathcal{C}[next(\sigma_{m-2})] - 1]$ such that $\Psi[s_{m-2}] \geq s_{m-1}$, while the latter is the position of the rightmost element in $\Psi[\mathcal{C}[\sigma_{m-2}], \mathcal{C}[next(\sigma_{m-2})] - 1]$ such that $\Psi[e_{m-2}] \leq e_{m-1}$. Algorithm 3.3.4 shows how to retrieve the value s_{m-2}, which is the value s' returned at the end of the algorithm. The algorithm employs two indexes *start*, *end*, which denotes the range of possible values for s_{m-2} at each iteration. This range is progressively refined by hinging upon the fact that $\Psi[\mathcal{C}[\sigma_{m-2}], \mathcal{C}[next(\sigma_{m-2})] - 1]$ is sorted. Specifically, the range is updated by looking at its middle element: if this falls outside $\{s_{m-1}, \dots, e_{m-1}\}$, then we update the indexes *start* and *end* in order to try moving the middle element inside $\{s_{m-1}, \dots, e_{m-1}\}$ (lines 10 and 7); conversely, if the middle element resides inside $\{s_{m-1}, \dots, e_{m-1}\}$, then it may be the leftmost element such that $\Psi[middle] \geq s_{m-1}$. Therefore, we save it as a candidate value for s_{m-2} (line 12) and we update *end* in order to look at elements on the left of the current candidate value for s_{m-2} (line 13). At the end of the algorithm, s' necessarily stores the position of the leftmost element such that $\Psi[s'] \geq \Psi[s_{m-1}]$: indeed, *start* is at most as big as the position of the first element of Ψ which is at least as big as s_{m-1} (i.e., the desired index s_{m-2}), while *end* is at least as small as the last element of Ψ which is smaller than s (i.e., $s_{m-2} - 1$); thus, s_{m-2} is necessarily the last position assigned to s' at line 12. In order to find e_{m-2}, we employ a dual procedure, which explores elements on the right of the candidate value whenever the middle element resides inside $\{s_{m-1}, \dots, e_{m-1}\}$. By applying the same procedure, we can compute indexes s_{m-3} and e_{m-3} from s_{m-2} and e_{m-2}, and so on, until all the m characters of the whole pattern are considered.

The algorithm we just described can be performed obliviously if the Ψ array is arranged in $|\Sigma|$ separate search trees, which are wrapped into an ODS as discussed for STBWT. Nevertheless, we also devise a way to perform a search over the whole Ψ array in order to gain the ability to pack Ψ within a single-rooted ODS. We first describe the modified search procedure which does not necessarily start from the range related to character σ in the Ψ array (i.e., $\Psi[\mathcal{C}[\sigma], \mathcal{C}[next(\sigma)] - 1]$), as ensured by lines 1 and 2 of Algorithm 3.3.4; then, we show how to adopt a single-rooted search tree to perform this search over Ψ. Algorithm 3.3.5 show the search procedure to compute the index s_{m-i}, which is done over all the Ψ array (first two lines of the algorithm). When the current *middle* element does not belong to the portion of the Ψ array corresponding to the current character σ, which is $\Psi[\mathcal{C}[\sigma], \mathcal{C}[next(\sigma)] - 1]$, the value of $\Psi[middle]$ is ignored: instead, the search proceeds in the left partition (i.e. left subtree) if $middle < \mathcal{C}[\sigma]$ (line 8) or in the right if $middle > \mathcal{C}[next(\sigma)] - 1$ (line 8). When middle falls in the right range, instead, the value of $\Psi[middle]$ is taken into account, and processed as in Algorithm 3.3.4: hence, if it falls in the right boundaries $[s, e]$ (lines 15 and 18) it is saved in s', and the search continues among the leftmost (respectively, rightmost) element during the search for s_{m-i} (respectively, e_{m-i}). The complexity of this implementation of backwards

Algorithm 3.3.4: Leftmost Ψ − SEARCH

```
Input: The $\Psi$ array of $T$
Input: The previous interval $s$ and $e$
Input: The current character $\sigma$
Input: The $\mathcal{C}$ array
Output: The new start of the inteval $s'$
// indices of the binary search
1  $st \leftarrow \mathcal{C}[\sigma]$
2  $end \leftarrow \mathcal{C}[next(\sigma)] - 1$
   // -1 means no matches, i.e.  failure
3  $s' \leftarrow -1$
4  while $start \leq end$ do
5      $middle \leftarrow st + \lfloor \frac{end-st}{2} \rfloor$
6      $psi \leftarrow \Psi[middle]$
       // try to find the interval of $\Psi$ pointing to $[s, e]$
7      if $psi > e$ then
8          $end \leftarrow middle - 1$
9      end
10     else if $psi < s$ then
11         $start \leftarrow middle + 1$
12     else
13         $s' \leftarrow middle$
           // leftmost search always goes to the left
14         $end \leftarrow middle - 1$
15     end
16 end
17 return $s'$
```

search is $\mathcal{O}(m \cdot \log N)$.

This intricate search criterion is devised to pack the Ψ array in a single-rooted binary tree, which is, generally speaking, a more traditional way to exploit ODS's.

While this algorithm exhibits a more complex control flow than the previous, it is possible to transform it in an oblivious one by resorting to the `ternary_operator` primitive. We will show the final complexity of this algorithm later, when explaining the structure of the ODS that hosts Ψ.

3.3.4 Binary search tree ODS

In Subsections 3.3.3 and 3.3.2, we explained how to use an ODS to achieve obliviousness for two of our algorithms, namely SA-Ψ and STBWT, without specifying how such an ODS should be structured. We only showed that an array that exhibits some kind of ordering may be arranged in a tree recreating the access pattern of a binary search, as depicted in Figure 3.6(a) and (b), that in fact serves as a way to turn both Ψ and occ_σ into search trees.

It turns out that the data structure we obtain via this process is ready to be put into an ORAM. In particular, we start the insertion from the root, and properly link each node to its children by assigning them a block id *bid*, which can be chosen akin to the numbers shaded in black in Figure 3.6(b), and a leaf id that is used to evict the children but is also kept in the parent node as a pointer, similarly to what we did for recursive position maps. In the basic construction, all the nodes are stored within the same ORAM, whose capacity must be big enough to host all of them. If we assume that each ORAM access introduces an overhead proportional to $\log N$, and that a root to leaf exploration requires the traversal of $\log N$ levels, the resulting complexity of a round of search would be:

$$\sum_{i=0}^{\log N} \log N = \mathcal{O}(\log^2 N)$$

We can achieve a better complexity by restructuring the ODS.

The access pattern of a binary search is fixed: exploration starts from the root, traversing all the levels in sequence until it reaches a leaf. In order to optimize the traversal, we may pack each level of the binary tree in a

Algorithm 3.3.5: Full leftmost Ψ – SEARCH

```
Input: The $\Psi$ array of $T$
Input: The previous interval $s$ and $e$
Input: The current character $\sigma$
Input: The $\mathcal{C}$ array
Output: The new start of the inteval $s'$
// indices of the binary search
1  $st \leftarrow 0$
2  $end \leftarrow |\Psi| - 1$
   // match boundaries
3  $s' \leftarrow -1$
   // indices of the SA matching $\sigma$
4  $range\_left \leftarrow \mathcal{C}[\sigma]$
5  $range\_right \leftarrow \mathcal{C}[next(\sigma)] - 1$
6  while $start \leq end$ do
7      $middle \leftarrow st + \lfloor \frac{end-st}{2} \rfloor$
       // try to find the interval [range_left, range_right]
8      if $middle > range\_right$ then
9          $end \leftarrow middle - 1$
10     end
11     else if $middle < range\_left$ then
12         $start \leftarrow middle + 1$
       // right range
13     else
14         $psi \leftarrow \Psi[middle]$
           // try to find the interval of $\Psi$ pointing to $[s, e]$
15         if $psi > e$ then
16             $end \leftarrow middle - 1$
17         end
18         else if $psi < s$ then
19             $start \leftarrow middle + 1$
20         else
21             $s' \leftarrow middle$
               // leftmost search always goes to the left
22             $end \leftarrow middle - 1$
23         end
24     end
25 end
26 return $s'$
```

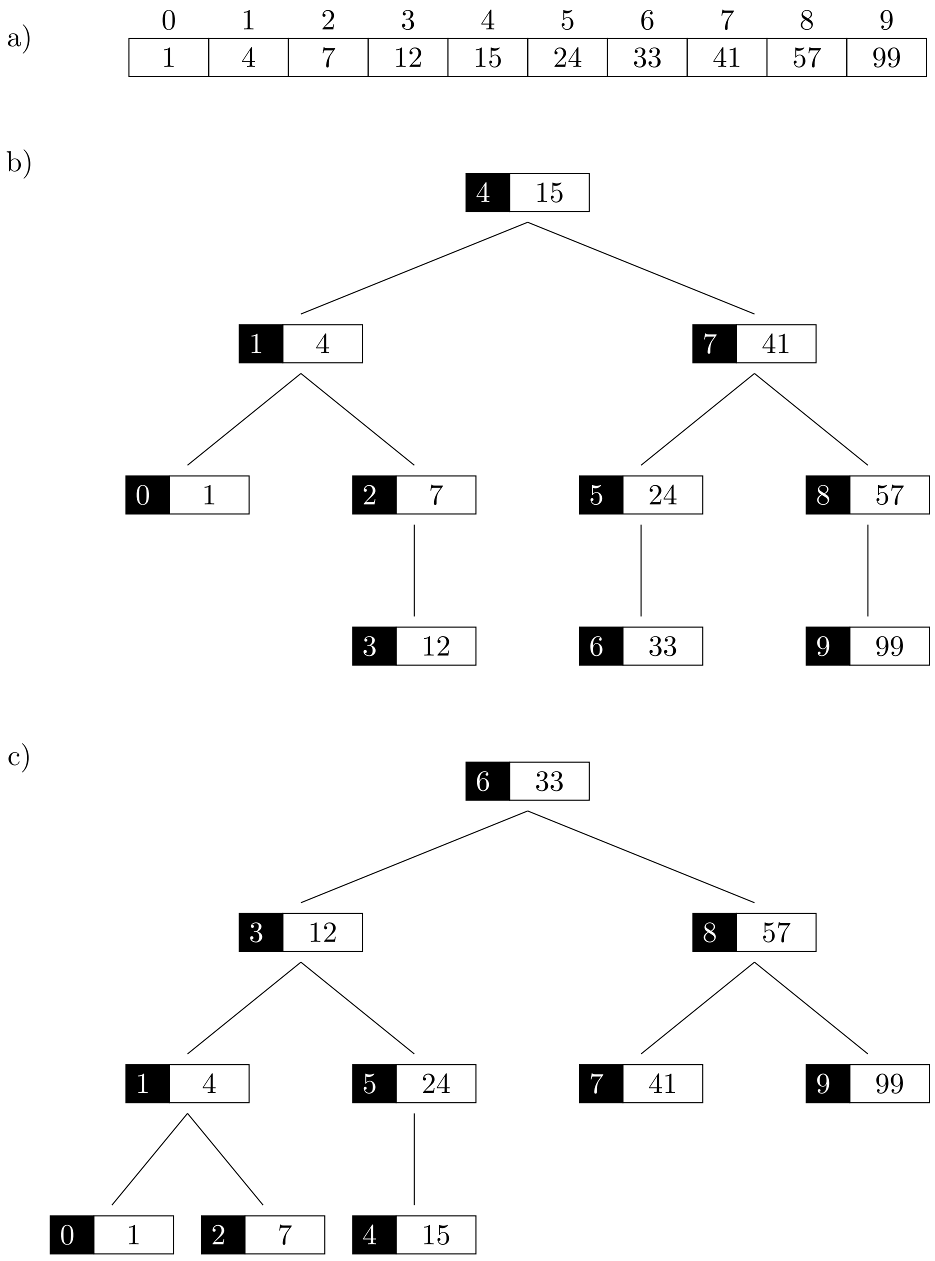

Figure 3.6: (a) A sorted array (b) a balanced binary search tree and (c) a complete balanced binary search tree

separate ORAM [18]: thus, ORAM_i will store the i-th level of the search tree, that in turn contains 2^i nodes assuming that levels are numbered from 0 (root) to $\log N$. The complexity we achieve is:

$$\sum_{i=0}^{\log N} \log 2^i = \sum_{i=0}^{\log N} i = \mathcal{O}(\frac{\log^2 N}{2})$$

Even if we improve only by a constant factor, this optimization turns out to have a strong impact on final performance, which is line with the results showed in previous works [18, 25]. As an example, the ODS for STBWT is reported in Figure 3.7: here, all the nodes in the same level of each of the $|\Sigma|$ trees are merged in a single ORAM and each level is stored in a separate ORAM.

To construct our ODS, we initialize the search tree level by level instead of performing a full binary tree insertion for each element. This is arguably the most efficient way to add nodes to the tree, since exactly N accesses to the ORAM are required. Nevertheless, this initialization strategy requires that the parent knows the correct leaf ids of its two children nodes, even if these have not been inserted into the ODS yet. In order to overcome this issue, we generate leaf ids via a deterministic Cryptographically Secure Pseudorandom Number Generator (CPRNG), such as AES in CTR Mode. More specifically, let Π_k be a CPRNG seeded with k that exposes a function Gen to produce random values and Init to generate the sequence from the beginning. Let GenSeed be a function that returns a random seed – on `x86_64`, the instruction `RDSEED` would serve the purpose. Algorithm 3.3.6 shows how a single level of the path can be initialized: it inserts one node at a time in each level, using $k2$ to generate its children pointers and $k1$ to determine the leaf corresponding to the path where it will be evicted. $k1$ is the seed that was used to fill the children pointers of the previous level, so that the newly inserted elements are correctly pointed by the ones in the previous level. The initialization of the block ids that complete an ORAM pointer is omitted, since reasonable block ids, such as the ones shaded in black in Figure 3.6(b), can be easily generated. $k2$ is returned by LevelInitialization, and is used by the ODS Construction procedure (Algorithm 3.3.7) to link two consecutive levels.

This way of instantiating an ODS does not leak any secret: in fact, even if the new nodes are explicitly assigned a path, they are just appended to the stash at insertion time. The paths that are evicted due to ORAM access are either random (Path ORAM) or deterministic (Ring and Circuit ORAM), and thus are not related at all to the path assigned to the inserted node.

Algorithm 3.3.6: Binary tree ODS LevelInitialization

```
Input: The current level $l$
Input: An array of $2^l$ search keys $key$
Input: The seed $k_1$ used for the previous level
Output: The seed $k_2$ used for the current level
1  $k_2 \leftarrow$ GenSeed()
2  $\Pi_{k_1}$.Init()
3  $\Pi_{k_2}$.Init()
4  for $i \leftarrow 0$ to $2^l - 1$ do
5      $node \leftarrow$ CreateEmptyNode()
6      $node.key \leftarrow key[i]$
       // construct the pointers to the children
7      $node.left.leaf\_id \leftarrow \Pi_{k_2}$.Gen()
8      $node.right.leaf\_id \leftarrow \Pi_{k_2}$.Gen()
       // generate and assign the leaf id
9      $ev\_leaf \leftarrow \Pi_{k_1}$.Gen()
10     $node.leaf\_id \leftarrow ev\_leaf$
11     Add node to the stash of ORAM$_l$ and perform the eviction
12 end
   // return the seed for the next level
13 return $k_2$
```

Nonetheless, the last level $L = \lfloor \log N \rfloor$ of the tree represents a problem for this algorithm. In fact, its nodes are scattered, and generating the correct leaf id via a CPRNG requires to go through the previous ones as well. In order to avoid leakage, it is necessary to perform dummy accesses as if all the nodes in $L - 1$ had two children, further decaying performance.

The solution we adopted is to build a *complete* binary search tree, in which all the nodes on the last level are compacted to the left, as in Figure 3.6(c).

In order to build a balanced tree that is also complete starting from a sorted array, it is necessary to carefully choose the index *root* of the original

Algorithm 3.3.7: Binary tree ODS CONSTRUCTION

```
Input: A list of levels of a search tree $lev$
// get number of levels
1 $L \leftarrow |\, lev \,|$
// initialize seed $k_1$
2 $k_1 \leftarrow$ GENSEED()
3 for $i \leftarrow 0$ to $L-1$ do
4     $k_2 \leftarrow$ LEVELINITIALIZATION($i, lev[i], k_1$)
5     $k_1 \leftarrow k_2$
6 end
```

array that will serve as a root of the complete tree. In particular, *root* splits the array A in two partitions, $[0, root-1]$, $[root+1, |A|-1]$ that need to contain the correct number of elements so that recursive application of the root selection for all the subtrees in lower levels yields a complete tree.

A *perfect* binary tree is a complete binary tree whose last level is full. It is easy to observe that any complete tree is contained within a perfect one. We will number the levels of a tree from 0 (root) to $L = \lfloor \log N \rfloor$, where N is the number of nodes in the tree. A perfect tree contains $2^{L+1}-1$ nodes, 2^L of which in the last level. Half of the nodes in the last level (2^{L-1}) will belong to the left subtree of the root, the other half to the right. A complete tree has $x \leq 2^L$ nodes on its last level. The last level of the left subtree will thus contain $k = min(x, 2^{L-1})$ elements. The left subtree, that starts at level 1, is complete up to level $L-1$, and thus contains $N_l = \lfloor 2^{L-2} \rfloor + k$ nodes (the floor function is applied in case $L = 1$). Hence, index N_l correctly partitions array A and can be chosen as a root for the tree. The same reasoning can be applied to choose the elements from subarrays $A[0, root-1]$ and $A[root+1, |A|-1]$ that must become the roots of, respectively, the left and right subtrees of the complete tree. Algorithm 3.3.8 sums up this function, while the numbers shaded in black in Figure 3.6(c) show the results of recursive application of the subroot selection.

The strategy we have just described allows to efficiently initialize the ODS. As a consequence, the sequence of elements inspected during the binary searches performed in Algorithms 3.3.2 and 3.3.5 must correspond to a path in a complete binary tree constructed by subroot selection procedure in

Algorithm 3.3.8: SubrootSelection

```
Input: The starting index of a partition $s$
Input: The length of a partition
Output: Index of the subroot relative to a 0-indexed array
1 if $len == 1$ then
2     return 0
3 end
4 $L \leftarrow \lfloor \log(len) \rfloor$
  // number of elements in the last level
5 $x \leftarrow len - (2^L - 1)$
6 $k \leftarrow min(x, 2^{L-1})$
7 return $s + \lfloor 2^{L-2} \rfloor + k$
```

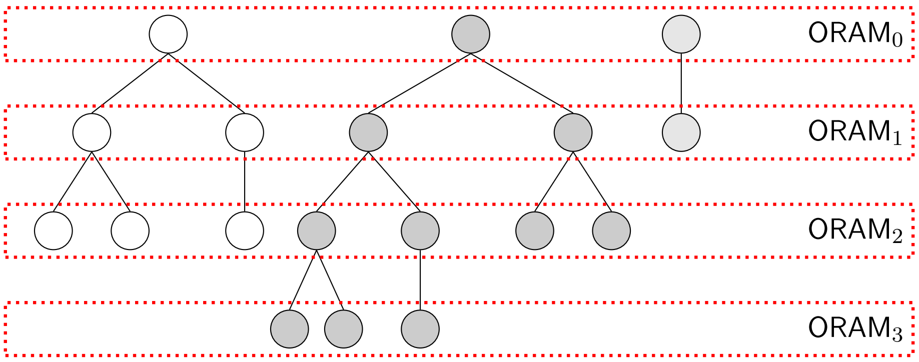

Figure 3.7: ODS of STBWT

Algorithm 3.3.8; this is equivalent to choosing the *middle* element (line 5 in Algorithm 3.3.2 and line 7 in Algorithm 3.3.5) by hinging upon the subroot selection procedure we have just described.

Now that we have introduced the structure of our ODS, we can evaluate how obliviuosness impacts the performance of SA-Ψ and STBWT. Once again, we adopt Circuit ORAM access time, $\mathcal{O}(B \cdot \log N)$. However, the blocks of the ORAM have a fixed structure, consisting of the search key (i.e., the corresponding entries of Ψ and occ_σ in, respectively, SA-Ψ and STBWT ODS) and the pointers to the two children. As a consequence, the access time becomes $\mathcal{O}(\log N)$, as the cost $\mathcal{O}(B)$ to obliviously copy the content of a block is equivalent to simply copying one integer and two pointers, which is obviously $O(1)$. Our ODS consists of $\log|T|$ levels, that the client traverses for each character of the pattern. Therefore, the overall complexity of these algorithms becomes:

$$m \cdot \sum_{i=0}^{\log|T|} (\log 2^i) = m \cdot \sum_{i=0}^{\log|T|} i \in \mathcal{O}(m \cdot \log^2 |T|)$$

thus adding a $\log |T|$ term to their initial complexity.

3.3.5 Security and leakage of the proposed solutions

The security of our solutions is based on three pillars: the trusted execution environment provided by Intel SGX, the confidentiality and integrity guarantees obtained by using AE ciphers and the obliviousness of the access patterns of the algorithms which is mainly guaranteed by ORAM. The full-text indices are computed by the client in a preprocessing stage and encrypted offline with an authenticated block cipher, in order to guarantee both their confidentiality and their integrity. Plaintext information within the binary files, such as the length of the text and the size of the input alphabet, is authenticated as well. Then, these encrypted indices are outsourced to the server (Figure 3.1). During the setup phase of ObSQRE, depending on the algorithm chosen by the client to perform the substring search queries, several ORAMs are instantiated and filled by the secure enclave with the content of the full-text indices, which are fetched from the untrusted storage and decrypted inside the enclave. The ORAMs use ephemeral random keys generated by `RDRND`, that are securely stored in the enclave and wiped as soon as the client tears down a session. This ensures that it is not possible to dump the content of the ORAM tree to replay a whole session, for example querying data structures for which an attacker doesn't own the keys.

All the sensitive indexing structures are packed within an ODS or a recursive ORAM: since they totally hide the logical access pattern of the search algorithms, it is not possible to infer any information about the content of the indices; furthermore, ORAMs allow to conceal the the similarity of subsequent queries, thus protecting the search patterns. Furthermore, the shape of the indexing structures, i.e. the number of levels of the ODS's or position map, is publicly known, so there is nothing to learn about it.

We now separately discuss the text indices of the corresponding substring search algorithms in order to analyze the information leaked by each of them.

In the BWT backwards search, we use an ORAM with recursive position map. The sample period R does not affect the security, but rather the tradeoff chosen by the client between computation and memory consumption, and it is assumed to be publicly known since it can be easily exfiltrated by the adversary from the execution of Algorithm 3.3.1 in the enclave. The overall leakage is limited to the length of the original text and the size of its alphabet.

In order to guarantee obliviousness, SA-Ψ performs a root to leaf access to the ORAMs composing its ODS for every query character. The last level may not be complete: however, L accesses to the ORAM must be done, because otherwise the adversary can infer that the binary search algorithm is visiting a shorter or longer path of the binary search tree. To overcome this issue without inserting dummy block in the last level of tree, namely ORAM_L, the algorithm performs an access to one of the dummy blocks (with $bid = \perp$) of ORAM_L when a node on level $L-1$ has no children. Thanks to this trick, SA-Ψ only leaks the length of the original string.

Conversely, STBWT introduces more challenges in this regard. The trees associated to each character may exhibit different depths, thus leaking information about the relative frequencies of characters. In this case, it is necessary to perform accesses from ORAM_0 to ORAM_L in order to conceal the character we are looking for. To avoid the leakage of character frequencies, it is possible to pad each search tree to the size of the one of the most recurring character, severely impacting memory usage. Nonetheless, we stick to the first approach, as we deem the information about the relative frequencies of characters not significant enough to motivate the performance overhead required to hide this leakage; however, we remark that we also develop a more safe version that only leaks the number of occurrences of the most frequent character.

Concerning the leakage of the length m of the pattern being searched in a query, it is easy to observe that all the backwards search algorithms perform a m iterations, thus the adversary can trivially infer this information. To mitigate this problem, we give the client the possibility to pad each query with a varying number of dummy characters placed at random. Dummy characters exhibit the same behavior of real ones but don't contribute to the update of the start and end indices that mark the matches.

The result of each query, that is the positions of all the occurrences of a pattern on the suffix array, is not revealed, as their limits are returned in encrypted form. In order to retrieve these indices, a suffix array is optionally enclosed within the index. The suffix array is packed in recursive ORAM, that stores res entries inside each block. The client can fetch a portion of the suffix array only at this granularity, however the number of chunks retrieved is proportional to the number of occurrences of the pattern found in the text. To this extent, ObSQRE allows the client to request additional chunks which are not related to the occurrences of the pattern, in order to hide the number of matches; the decision about the number of dummy chunks to be retrieved is delegated to the client, which can choose the trade-off between the information leakage and latency of a query depending on the application scenario.

Lastly, we recall that for Circuit and Path ORAMs, it is possible to overlap the ORAM tree with a Merkle tree nodes, in turn guaranteeing the integrity and freshness of the buckets and avoiding replay attacks. Therefore, when the full-text indices are wrapped into either Circuit or Path ORAM, our solutions are secure even against malicious adversaries as well. Conversely, when Ring ORAM is employed, the security and leakage guarantees of our solutions are provided only only against a passive attacker, who audits the application without direct intervention.

Chapter 4

Implementation

In this chapter we carefully review the *remote attestation* process, that allows a remote party to establish the authenticity of an enclave allocated on a distinct and potentially malicious machine. The scenario envisioned by Intel SGX is mainly oriented to Digital Rights Management (DRM) or similar purposes. In this case, the user of an enclave is a client who wishes to access protected contents distributed by a provider that enforces specific terms of use. When a client wants to ask for some resources, she has to execute a secure enclave developed by the provider in order to safely deliver and handle such copyrighted data. Before sending contents over the network, the servers of the provider need to ensure that the enclave that was instantiated on the client is the right one: the remote attestation step guarantees its authenticity, also performing an authenticated Diffie-Hellman Key Exchange (DHKE) to establish a secure channel that will be employed for further communications. In this context, the client represents the untrusted entity, who potentially wishes to abuse some digital contents, and the remote attestation is carried out by the content provider, that corresponds to the ISV that developed the enclave. Intel grants access to the IAS, which needs to be queried in order to verify the authenticity of the attestation proof generated by the enclave, only to the ISV, strongly suggesting that indeed Intel SGX was mainly conceived for DRM.

The case of cloud computing is totally different though. In particular, a client who wishes to process a huge amount of data outsources the computation to an untrusted remote server with bigger storage and computational capabilities. When the dataset of the client is sensitive, she may exploit a secure enclave to protect its content during the computations: hence, the client wishes to perform remote attestation to establish the authenticity of an enclave before trusting it. We point out that the user of a specific enclave not necessarily corresponds to the original enclave developer, and it is likely unwilling to undergo all the annoying steps needed to gain access to the IAS, which includes requesting an X.509 certificate from a trusted certification authority.

In this regard, we show how the usual flow of the attestation procedure can be modified in order to achieve this purpose, i.e. allowing a client to attest an enclave that was developed by a third party ISV and is run on a remote server. Since the ISV still needs to intercede in the access to the IAS, we also show how to prevent an unfaithful ISV from colluding with a cloud service provider to cheat during the remote attestation. Surprisingly enough, all the previous works that rely on Intel SGX in the cloud setting did not consider these pitfalls at all. Section 4.1 goes through the original protocol devised by Intel and then explains how it can be adapted to our needs.

Section 4.2 gives a rapid overview of the Intel SGX programming model, highlighting the limitations of this technology and some workarounds to circumvent them. Section 4.3 provides a high level overview of the several parts composing ObSQRE, specifically showing the sequence of operations that a client performs to query a full-text index of his own, including the several data exchanges required by remote attestation.

4.1 Intel SGX remote attestation overview

As discussed in Section 2.1, *remote attestation* allows a remote party to verify that the secure enclave was correctly instantiated on the remote untrusted machine and to establish a secure channel via a key exchange. We now provide an high level overview of the remote attestation protocol devised by Intel, highlighting some hindrances which harden its employment in our cloud-computing scenario. Remote attestation first performs a DHKE to derive a common key between the enclave and the ISV, which is then used to instantiate a secure channel; then, it generates a proof of authenticity of an enclave, referred to as *quote*, which can be later sent by the ISV to the IAS, obtaining an *attestation report* which specifies the soundness of the quote. Specifically, the quote is signed using a per-CPU key, called *attestation key*, that is only accessible to a special enclave authored by Intel, the Quoting Enclave (QE). The usage of a privileged enclave prevents any other entity from forging rogue quote signatures, and thus guarantees that the machine is equipped with legitimate and updated Intel SGX hardware. The consequence is that an attacker cannot simulate the steps performed by Intel SGX to perform remote attestation.

The quote is composed of several fields that allow to certify the correct setup of an enclave and the identity of its developer, the ISV. The former function is fulfilled by the `MRENCLAVE`, the latter by the `MRSIGNER`, both derived using Secure Hash Algorithm 2 (SHA-2).

The `MRENCLAVE`, which stands for 'enclave measurement', is generated during the setup of the enclave. The CPU exploits specific Intel SGX instruction to assign EPC frames to an enclave and initialize their content. These instructions update a SHA-2 hash that testifies both the sequence of operations and the data that were used to build the enclave [12]. The hash is kept in the EPCM and updated by microcode, so that it cannot be tampered with. `MRENCLAVE` includes both the code and the data of the application running in the enclave; therefore, if the `MRENCLAVE` matches with the one which identifies the enclave, it means that the machine did not cheat in the process of creating the enclave and thus the remote user of the enclave can trust the application running inside it.

On the other hand, the `MRSIGNER` is employed to identify the ISV. Specifically, each ISV employs an RSA keypair with public exponent 3 to sign its enclaves; the `MRSIGNER` is the SHA-2 hash of the modulus related to the keypair, which must be a valid 3072-bits RSA modulus. The signature appended to the enclave shared object is checked by Intel SGX after terminating its setup. The presence of `MRSIGNER` in the quote strictly binds each enclave to its developer.

When the remote party receives the quote along with its signature, it can query the IAS to verify that the process was successful.

The IAS can verify the attestation signatures generated by any Intel CPU that features Intel SGX. However, this is potentially harmful for privacy, since IAS would be able to track the software executing on each of its CPUs. In order to circumvent this drawback, Intel adopts a group signature scheme, called Enhanced Privacy ID (EPID). Each CPU is assigned to an EPID group containing a huge number of CPUs, in order to blur their identities. The public key of the EPID group can verify the signatures generated by all the processors in the group: this way, the CPU vendor cannot exactly know which piece of hardware actually generated the signature. IAS blacklists attestation signatures coming from compromised platforms, which are included in the Signature Revocation List (SigRL).

There are two caveats in the attestation procedure which hinders its adoption by entities different from the ISV. First, the DHKE which is performed at the beginning relies on a keypair $\langle ISV_{pub}, ISV_{priv} \rangle$, whose public key must be embedded in the enclave and included in the data authenticated in the `MRENCLAVE`, while the private key is retained by the ISV in order to reliably perform the DHKE. Therefore, the private key must be shared with all the entities willing to remotely attest the enclave, which may be a serious concern for the ISV. The second caveat is that Intel only allows the ISV to access IAS. Indeed, an ISV needs to perform several steps in order to be granted access to IAS:

1. first, it has to obtain an X.509 certificate from a trsuted certification authority, (though the certificate can be self-signed for testing purposes);
2. then, it subscribes to Intel Development Services;
3. finally, it waits for the grant by Intel, which sends a Service Provider ID (SPID) via e-mail.

The ISV uses the certificate to perform mutual TLS authentication with the IAS. The end user of the enclave is likely not willing to undergo all these steps in order to utilize the application running in the enclave, preventing their adoption in our cloud computing scenario. In conclusion, it is rather clear that the remote attestation procedure cannot be straightforwardly applied to our cloud computing scenario, hindering the adoption of secure enclaves for outsourced computation. Therefore, we deeply analyzed the Intel SGX attestation procedure to obtain a revised procedure which can be carried on

Table 4.1: Format of the first message

Message			**Decription**
Message m_1	Ga	Ga_x Ga_y	Ephemeral P-256 public key
	EPID Group Id		To verify the signature of the QE

by several entities which are simple users (not owners) of the enclave without undergoing annoying steps such as requiring digital certificates. Quite surprisingly, this issue has never been analyzed in existing works presenting Intel SGX based applications. We now present our analysis and our revised attestation procedure, starting with a detailed description of the remote attestation procedure devised by Intel.

4.1.1 Client-to-Server attestation

The attestation and key exchange protocol devised by Intel [10] allows the ISV to authenticate the enclave running on the client machine (DRM scenario). Since the client is untrusted, we assume that it will play the role of a Man-in-the-middle (MITM) during the execution of the key exchange. Moreover, we assume that the part of the computation executed by the enclave strictly follows the protocol, which is true if the ISV implemented it properly. Throughout this subsection, we use $CMAC_k(m)$ to refer to the AES-128 Cipher-based MAC (CMAC) of m with key k.

Before provisioning secrets to an enclave, the ISV asks the enclave to attest itself. In the process, a key exchange establishes a secure channel that can be used for further communications.

Phase 1 The client fetches its EPID Group Id, that is used by IAS to verify the quote signatures produced by a group of CPUs. It initializes an attestation context within the enclave that holds the intermediate results of the key exchange. The attestation context contains the public key of the ISV ISV_{pub}, which is a point on the elliptic curve P-256. To ensure that the correct public key is used, it can be hardcoded within the enclave image: as such, it will become part of the `MRENCLAVE`. Finally, the client generates an ephemeral keypair over the curve P-256: we will refer to the public portion of the key as $Ga = (Ga_x, Ga_y)$. The message sent to the ISV resembles the format of Table 4.1.

Table 4.2: Format of the second message

Message			**Decription**
Message $m_{2,A}$	Gb	Gb_x Gb_y	Ephemeral P-256 public key
	Parameters		Contains the SPID of the ISV
	Sig_{isv}		Thwarts MITM and authenticates the ISV
Message $m_{2,B}$	SigRL		Signature Revocation List
$CMAC_{SMK}(m_{2,A})$			MAC to verify integrity of message $m_{2,A}$

Phase 2 The ISV server queries the IAS to retrieve the SigRL of the client's EPID group. The signature generated by the QE changes according to the SigRL, which blacklist compromised CPUs for which secret keys have been leaked. Then, it generates an ephemeral keypair over P-256, with public key $Gb = (Gb_x, Gb_y)$ and private key Gb_p. It can then complete the DHKE evaluating $Gab = Gb_p * Ga = (Gab_x, Gab_y)$, where $*$ denotes the product of two points over P-256 elliptic curve. It then computes the Key Derivation Key (KDK) as $CMAC_0(Gab_x)$. The KDK will serve as the shared secret between the ISV and the enclave.

The KDK is employed to compute the Sigma Key (SMK) as:

$$CMAC_{KDK}(0x01 \,||\, "SMK" \,||\, 0x80 \,||\, 0x00)$$

The SMK, that can be derived within the enclave as well, is used to generate the MACs of subsequent messages, in order to avoid tampering.

In the DRM scenario, the enclave wants to establish the identity of the remote party. In order to do that, the ISV uses its private key ISV_{priv} to sign the message $Gb||Ga$, yielding Sig_{isv}. The aim of this step is twofold: first, it authenticates the ISV server to the enclave, since the ISV is the only owner of ISV_{priv}. Furthermore, it allows the enclave to verify both that the server received the right ephemeral public key Ga and that the ephemeral public Gb received by the enclave has not been tampered with, thus, thwarting any MITM attack from the untrusted client.

Finally, the ISV includes in the message other metadata, including its SPID. The format of this message is reported in Table 4.2.

Phase 3 Intel SGX API provides routines to perform the following operations within the enclave:

Table 4.3: Format of the third message

Message			**Decription**
Message m_3	Ga	Ga_x Ga_y	Ephemeral P-256 public key
	Quote		Quote generated by the QE
$CMAC_{SMK}(m_3)$			MAC to verify integrity of message 3

1. derive KDK and SMK using the ephemeral public key *Gb* received from the ISV in message m_2;

2. verify Sig_{isv} with ISV_{pub};

3. verify the CMAC of the message $m_{2,A}$.

If all these checks are successful, it requests the quote to the QE, and assembles a message as reported in Table 4.3. Since this message is authenticated with SMK, the ISV can verify that the *Ga* it used to derive the KDK was legitimate. The quote, besides `MRENCLAVE` and `MRSIGNER`, includes another field, called `REPORTDATA`, which contains the SHA-2 hash of *Ga* || *Gb* || VK, where the Verification Key (VK) is computed as $CMAC_{KDK}(0x01||"VK"||0x00||0x80||0x00)$. `REPORTDATA` play a pivotal role in our revision of the attestation procedure to make it suitable for our scenario.

Phase 4 The ISV needs to implement these checks:

1. check correctness of the CMAC;

2. verify that the *Ga* in the message matches the one used to derive the KDK;

3. evaluate VK and check the content of the `REPORTDATA`, to verify that the enclave received the correct *Gb*.

If these checks are successful, the ISV can submit the quote to the IAS. A nonce may be optionally submitted to the IAS via its HTTP API, an important feature that will come in hand in our scenario [9]. IAS answers "OK" if the attestation signature was correctly verified, appending the content of the request submitted by the ISV. IAS signs the request submitted by the ISV, so that she can verify that the content of its request was correctly received. It then may check the nonce for freshness, if it was previously included in the verification request. If all these security checks are successful, the ISV extracts the `MRSIGNER` and `MRENCLAVE` from the quote and checks it against the ones it authored. At this point, both the enclave and the ISV can derive the Session Key (SK) and Master Key (MK):

$$\begin{cases} SK = CMAC_{KDK}(0x01 \,||\, "SK" \,||\, 0x00 \,||\, 0x80 \,||\, 0x01) \\ MK = CMAC_{KDK}(0x01 \,||\, "MK" \,||\, 0x00 \,||\, 0x80 \,||\, 0x01) \end{cases}$$

that can be used to encrypt subsequent data exchanges, thus concluding the remote attestation procedure.

4.1.2 Security against MITM attacks in our scenario

In phase 2 of the remote attestation procedure, the pair of ephemeral public keys Gb, Ga generated for the attestation by the ISV and the enclave are signed with the private key ISV_{priv} of the ISV: this prevents MITM attacks from the untrusted machine where the enclaves resides, as this machine cannot produce a valid signature where the ephemeral public key of the ISV (i.e., Gb) is replaced by a key Gc chosen by the adversary, which is needed to perform a successful MITM attack. Nonetheless, we have already discussed that the need to know ISV_{priv} is a severe hindrance for the adoption of the attestation procedure in our could computing scenario. In this case, an entity (referred to as user or client) rents a untrusted server in the cloud and instantiates an enclave developed by a third party, the ISV. Even though the ISV is not trusted, the client may inspect the enclave binary to verify that it conforms to the specifications. On the other hand, remote attestation is needed to prove that the server actually instantiated the correct enclave. The ISV supposedly wants all the clients to use the enclave without having to recompile it with a distinct hardcoded public key for each user; furthermore, we assume that the ISV is not willing to share its private key ISV_{priv} with all the clients. In this case the attestation procedure we introduced earlier represents an obstacle to the wide adoption of enclave based applications.

Luckily, we observe that signing the pair of ephemeral keys Gb, Ga is not strictly required to prevent MITM attacks. To show this, we consider a remote attestation procedure where, in Phase 3, the enclave does not verify the signature Sig_{isv}. We suppose that the untrusted server in our cloud scenario tries to act as a MITM between the enclave and the client. In order to break the secure channel created via a DHKE, a MITM performs two separate key exchanges with the communicating parties, using its own public key Gc, for which he knows the private key Gc_p. The adversary replaces Ga in message 1 with Gc, and then generates a shared secret G_{bc} with the ISV as soon as it receives Gb. Then, it replaces Gb in message $m_{2,A}$ with Gc and forward the message to the enclave as well, establishing another secure channel based on the shared key G_{ac}. From now on, all the CMACs can be substituted, delivering a successful attack.

However, the attack can be easily detected by the client. This is due to the fact that the quote and its signature cannot be forged by a MITM, since they are generated by the trusted platform. The impossibility to forge enclave quotes derives from the fact that the QE is the only entity that can derive the attestation key that produces the signature. Therefore, a MITM cannot modify the `REPORTDATA` field of the quote, that contains the SHA-2 hash of $Ga \,||\, Gc \,||\, VK_{a,c}$. Ga is the public key generated by the enclave, while Gc is the public key employed by the attacker as a MITM. On the other hand, the client expects to find in `REPORTDATA` the hash of $Gc \,||\, Gb \,||\, VK_{c,b}$, that will mismatch with the one in the quote. `REPORTDATA` provides already full protection against a MITM attack and also guarantees freshness of the quote, hence preventing quote replaying attacks, as it contains an ephemeral public key generated by the ISV.

Even though not officially documented, we argue that the usage of the ISV keypair is only necessary to authenticate the ISV to the enclave. In fact, if not used, any remote party would be able to perform a key exchange with the enclave. In a scenario such a DRM, each ISV is interested in guaranteeing that its enclave is not exploited by third parties to distribute protected material. Conversely, in our scenario, the enclave is not interested in authenticating the entity performing the attestation: as we will see in the description of our revised attestation procedure, an entity still needs to interact with the ISV in order to successfully perform attestation, thus preventing unauthorized entities from reliably attesting the enclave.

4.1.3 Server-to-Client attestation

Since we acknowledge that the verification of the Sig_{isv} signature in phase 2 of the attestation procedure is not required in our scenario, we can skip this step. To this extent, we decide to avoid the definition of a custom attestation protocol since it would require to re-write some parts of Intel SGX API, which currently allows to perform the whole attestation procedure with few calls to specific routines that hide to the developer all the details about remote attestation. Conversely, in order to ensure the compatibility with the existing protocol, we mandate that the ISV generates a random P-256 keypair $\langle ISV_{pub}, ISV_{priv} \rangle$, whose public key is embedded in the enclave as part of its data, while the private key is sent to the authorized users. Nevertheless, as Sig_{isv} is actually useless in our revised attestation procedure, the ISV would be no longer concerned in case ISV_{priv} is publicly leaked (indeed ISV_{priv} may even be publicly shared by the ISV with no security issues). Although this solution may appear not elegant, it allows us to still rely on the existing Intel SGX API, which are likely to be severely scrutinized to assess their reliability and security and which are already included in the TCB of Intel SGX.

The other problem of our revised attestation procedure is that the client cannot perform mutual TLS authentication with IAS, hence she is prevented from verifying quote signatures. In fact, it is reasonable to assume that in the general case a user is not willing to request a certificate from a Certificate Authority and to subscribe to IAS. In this case, the ISV that developed the enclave can intercede between the client and the IAS. When performing remote attestation, the client sends the quote to the ISV along with a nonce, that will become part of the request submitted to the IAS. The ISV then forwards the response it receives to the client, that can verify its freshness thanks to the nonce. We remark that this solution does not require the user to trust the ISV, and it works also in the case the ISV colludes with the cloud server hosting the enclave itself. We don't implement a separate server to forward the requests to the IAS, but perform all the required security checks to ensure freshness, validating our approach. The overall picture is depicted in Figure 4.1.

We remark that our remote attestation procedure does not impact the

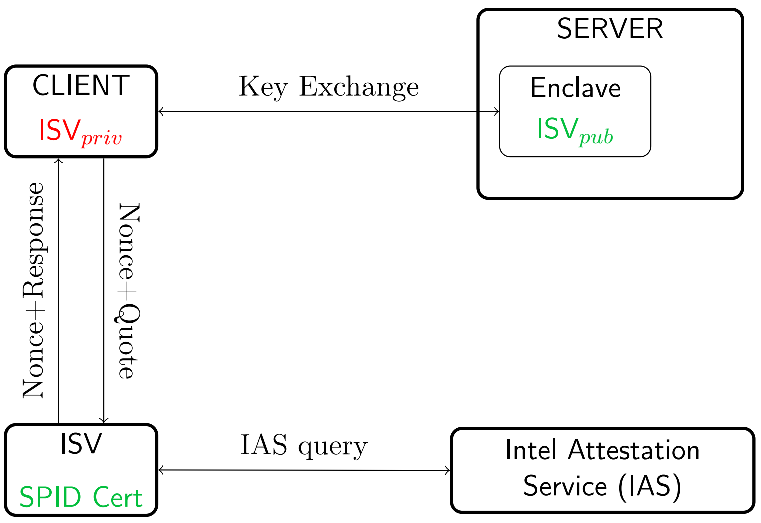

Figure 4.1: Overview of server-to-client attestation

security of our application. In fact, the indices used to perform substring search are encrypted using a symmetric key only owned by one client. Once the secure channel is established, it is employed to send this key to our enclave, which then decrypts the content of the index and allocates it in memory using either an encrypted ORAM or ODS. An attacker cannot decrypt the index of another client unless she derives the proper keys by other means. Furthermore, even though she managed to take over an initialized session of another client, she would not be able to perform any query, since they are encrypted using Intel AE with the SK, that is only known to the legitimate client.

4.2 Enclave programming model

Intel SGX enclaves are distributed as shared objects that are attached to the virtual address space of a process upon request. The setup of an enclave requires the usage of special instructions to allocate and fill EPC frames, computing the `MRENCLAVE`. The function `sgx_create_enclave` fulfills this purpose, returning the id `eid` of the instantiated enclave in case of success. `sgx_destroy_enclave` deallocates the enclave associated to a certain `eid`.

The code running on the untrusted part of the system may jump into an enclave only at predefined locations, established by the enclave developer.

The interface of an enclave must be in C and is declared in a special file with extension `.edl` [11]. The `.edl` file allows to declare both the methods that can be called by untrusted software, usually named `ecalls`, and the untrusted methods that may be invoked from within the enclave, named `ocalls`. Its syntax is similar to C, although it introduces some special constructs that decorate pointer arguments. Enclaves implement strict policies regarding pointers: in fact, a data buffer residing in unprotected memory locations may be accidentally modified by the enclave, leaking secret information. The Intel SGX toolchain automatically generates edger routines that move data buffers between the enclave and unprotected memory. This implies that the dimension of data buffer must be known a priori in order to move the correct amount of information. If a pointer is decorated with the keyword `[in]`, its content is dumped to a buffer stored within the enclave. Thus, the enclave can safely modify it, even writing secrets, since the changes do not apply to the original data buffer. A pointer decorated with `[out]` is initially allocated within the enclave, and its content is moved in the memory location at the end of the execution of the function. These two behaviours are combined if the developer specifies `[in,out]`.

On the other hand, when a pointer is declared as `[user_check]`, none of the operations we described is performed, and any sanity check is discretional. A `[user_check]` pointer is needed, for example, when allocating untrusted memory to store the ORAM trees. In fact, calling a `malloc` or `new` from within the enclave allocates protected memory, which is a very limited resource. `malloc` must be wrapped inside an `ocall` in order to allocate untrusted memory. Enclave developers can use specific methods offered by the SDK, namely `sgx_is_outside_enclave` to ensure that the pointer returned by the untrusted host refers to a valid memory location. In fact, no one prevents it from returning a pointer that refers to virtual addresses mapped to the EPC, thus causing the enclave to overwrite its own memory. This is why `[user_check]` pointers should be used only when strictly necessary.

All the functions declared in the `.edl` must return `void`. When invoked from the untrusted code, their prototype implicitly changes: in fact their first parameter must be the enclave id, followed by the other parameters declared in the `.edl` and they return an integer, which either signals successful invocation of an enclave function or failure (due, for example, to an incorrect initialization of the enclave).

In order to allow the inspection of an enclave, all the external libraries it uses must be statically linked in the enclave image. In fact, no one would prevent an untrusted OS from loading a compromised version of a library, and code running within the enclave can only invoke untrusted methods declared as `ocalls` anyways. It is not always straightforward to port libraries, e.g. cryptographic ones, for enclave use. Some libraries offer optimized versions of algorithms that leverage `x86_64` extensions, such as `AVX` and `SSE`. In order to choose the right implementation, most of them resort to the `CPUID` instruction, which is forbidden in enclave mode and causes a `SIGILL`. This was the case of `wolfCrypt`, a cryptographic library that we employ in our work and that is discussed later. Thus, it is often necessary to apply slight modifications to the original sources to account for the limitations of enclaves, that also encompass perform I/O, multithreading and some advanced functionalities normally provided by C/C++ [11].

The last major limitation of Intel SGX is that they don't allow to perform *system calls* as they imply jumping to kernel code, that is not included in the TCB according to the threat model adopted so far. In order to execute the routines of the OS it's necessary to invoke them via an `ocall`, and then return the results to the enclave. Before using them, the enclave should carry out several sanity checks in order to establish whether or not they look legitimate: ultimately, the enclave cannot rely on the operations performed by the OS as it may deliberately return malformed values to interfere with its execution or trigger vulnerabilities. Since rewriting wrappers for `syscalls` may be a tedious and lengthy task, library OS's such as *Haven* or *Graphene-SGX* [42] already provide these facilities. In ObSQRE we only need to read the contents of a full-text index from a binary file or allocate untrusted memory to host the ORAM tree, hence we prefer to implement these few functionalities by ourselves.

4.3 Implementation details

The functionalities of our enclave are exposed via a HTTP API that allows a client to perform remote attestation and substring search. All the server side of the application is developed in C/C++, while the client is written in Python.

We decouple the portion of the server that manages the connections from the implementation of the HTTP API. The class `multi_server` implements a basic multithreaded server that listens on a port and dispatches requests to worker threads. We adopt the `asio` library for networking because of its ease of use, but we do not exploit its full potential that relies in its *asynchronous model*. When used asynchronously, `asio`[1] serves many requests concurrently on the same thread because it does not hang on socket read/writes. All the operations are put in a queue along with a callback that is executed when network I/O has completed. While this technique allows to design servers that can handle thousand of connections with minimum latency, our application doesn't stress on networking performance. Therefore we opted for the more traditional and simple synchronous model, that waits for network I/O to complete before executing the next operation.

`multi_server` is an *abstract class*, and thus it cannot be instantiated. Its only function is accepting connections: any class that derives from it must implement the virtual method `void process_request(socket&)`, that serves the requests according to an agreed protocol. The reason behind this choice is that `multi_server` may come in hand when fast prototyping other remote protocols, saving us from re-implementing the networking part.

`subtol_server` implements the bare minimum part of HTTP needed to run our protocol. We decide to implement the necessary features ourselves because HTTP is a relatively simple protocol. We implement only a few HTTP headers, and perform some sanity checks on them:

- `Host`, that is mandated by the standard and contains the *domain name* of the server;
- `Content-Lenght`, the length in bytes of the HTTP body in case of `POST` requests;
- `Cookie`, that contains the identifier of a session of our application.

Even though we don't have in mind to firmly comply to the standard, we find out that our basic implementation works with all the clients we tried, i.e. cURL, Mozilla Firefox and the Python Requests library.

[1]think-async.com/Asio/ contains thorough reference of this library

We define a series of API calls that must be called in a certain order to establish a session. The requested method is encoded in the URI of the HTTP request and exploits cookies to keep track of a session. The eventual parameters and results are exchanged in JavaScript Object Notation (JSON) due to its flexibility and wide availability of libraries to handle it.

The `/start_session` call initializes a session and returns a random `session-id` cookie made of 32 alphanumerical characters. It performs the Phase 1 of the remote attestation protocol described in Subsection 4.1.1 as well. The cookie must be included in subsequent requests in order to update the state of a session. The `/attestation/1` and `/attestation/2` are necessary to complete the creation of the secure channel. The client posts Message 2 in the body of the `/attestation/1` request, and receives Message 3 in the response. At this point, the client can query the IAS to establish whether or not the enclave is trustworthy. She informs the server about the outcome via `/attestation/2`, which contains fixed ack/nack string encrypted with the Session Key (SK) to either proceed with the protocol or abort it. In our implementation, we don't implement the whole infrastructure depicted in Figure 4.1, but rather embed the ISV within the client. This decision was driven by the fact that splitting the ISV into a separate entity is just a programming task, that does not add any significant insights to the core of our work.

From there on, all subsequent communications are encrypted or authenticated with the SK and are thus protected against any eavesdropper. Moreover, since the attestation and key exchange protocols are secure against a MITM attacker even in the server-to-client scenario, the client has a strong guarantee that his SK is only known by the enclave.

We don't use HTTPS for our prototype, but we argue that it would not add any security margin to our application. In fact the untrusted server hosting the enclave may try a MITM attack at every moment. Since Intel SGX attestation provides protection against a local MITM, this holds for an adversary that tries to perform an attack over the network as well. Even if our application uses cookies that are sent in plaintext, an attacker would not be able to take over a session and perform queries on behalf of the real client. In fact, each command issued to the enclave is either encrypted or authenticated via the SK, that cannot be derived by any third party.

Although an attacker can take over a session at the beginning of a key exchange, he doesn't own the cryptographic key to decrypt the dataset of other client, and hence cannot query it. We notice that the malicious server can replay older queries, as well as a remote attacker that takes over a session. However, none of them can learn any information since the results are encrypted.

4.3.1 Oblivious primitives

We implement several ORAM protocols and substring search algorithms, and allow to freely combine them. With the `/configure` method, the client can select the desired ORAM protocol and its parameters. Since this information is public and can be easily inferred by the server, we only authenticate the configuration parameters.

All the oblivious primitives, save for the ODS implementing the binary search tree, are part of the `libobl`, which can be used for other enclaves as well. It must be compiled as a *static library* to be linked against an enclave image.

All of the ORAMs derive from an abstract class, `tree_oram`, that defines their interface. It exposes the following pure virtual methods:

- `access` wraps together block fetch and stash eviction;
- `access_r` and `access_w` correspond to the READ and WRITE methods of Algorithm 2.2.9 and come in hand when implementing both the recursive position map and the ODS, since the content of a fetched block containing ORAM pointers can be modified with the updated leaf ids before it is written back;
- `write` is useful for Algorithm 3.3.6 to skip path fetch when initializing the content of an ODS.

We can exploit *polymorphism* to allow the user to choose the preferred ORAM implementation.

Both the ODS and recursion require the instantiation of several ORAMs of varying size. In order to easily implement this, we employ the *abstract factory pattern*: each ORAM is associated to an *allocator* that instantiates ORAMs of a certain type and size. The classes implementing the recursive position map and the ODS are fed with the allocator, and invoke its method `spawn` to build a new ORAM object. `spawn` returns a pointer to a generic `tree_oram`, that hides the underlying implementation but exposes all the required methods thanks to *polymorphism*.

Among the ORAM implementations, there are singly oblivious versions of Path and Ring ORAM in order to evaluate the impact of doubly obliviousness on performance. Furthermore, `linear_oram` implements a simple oblivious primitive through linear scans over an array. It is useful to understand when ORAMs start to achieve better performance with respect to a trivial solution. `linear_oram` does not encrypt data, and thus is safe to use only if totally allocated within the enclave, as if it was a stash only.

We thought to another use for `linear_oram`. In fact, ODS-based schemes allocate very small ORAMs in the upper levels of the tree. Hence, we define a custom allocator that chooses a `linear_oram` instead of a `tree_oram` if the number of elements is below the break even.

The content of Circuit and Path ORAM is encrypted and authenticated using AES-128 in GCM, using an ephemeral random key generated via the instruction `RDRND`, available on most `x86_64` platforms. We tested several cryptographic libraries, finding out that `wolfCrypt` (which is a part of *wolf-SSL*[2]) achieves execution times that are by far better than its competitors, namely OpenSSL and Intel Integrated Performance Primitives (Intel IPP) library. In particular, the cryptographic part of Intel IPP is available in the Intel SGX SDK if it's built with the pre-compiled optimized libraries, otherwise an implementation based on OpenSSL is provided. `wolfCrypt` was initially considered because its low memory footprint suits the limited size of the EPC, but it manages to achieve outstanding performance thanks to the exploitation of AES New Instructions (AES-NI), that are specific opcodes introduced by Intel to speedup the computation of AES. Surprisingly, Intel IPP is better than `wolfCrypt` for AES in CTR Mode, since it exposes a more convenient API that allows to minimize the number of calls to the libray. Hence, Intel IPP is our choice for Ring ORAM.

[2]www.wolfssl.com/ links to the full documentation of both the SSL/TLS suite and the cryptographic library

Table 4.4: Format of the binary file containing a full-text index

Field	**Size (bytes)**	**Description**
Algorithm	8	0 or 4: SA-Ψ 1 or 5: STBWT 2 or 6: BWT
String length N	8	Length of the original text
Alphabet size	8	Size of the alphabet $\|\Sigma\|$
Size of integer I	8	Constant, 4
PBKDF2 salt size $size_s$	8	Size of PBKDF2 salt
AES-GCM IV	12	IV for AES (according to NIST)
PBKDF2 salt	$size_s$	Salt for PBKDF2
Sample period R	8	(only BWT) Sample period employed to build M_R
Character bits	8	(only BWT) Number of bits to encode a BWT character
Sample size	8	(only BWT) Size of an entry of M_R
Suffix array	$\sim (N+1) \cdot I$	Suffix array (optional)
$\mathcal{C}$ array	$\sim (\|\Sigma\| \cdot I)$	$\mathcal{C}$ array for backward search
Index	$\sim (N+1) \cdot I$	SA-Ψ: Ψ array STBWT: arrays of occurrences BWT: samples of matrix M
AES-GCM MAC	16	MAC of the whole file

4.3.2 Session setup, query and tear down

After choosing the preferred ORAM primitive, the client specifies the name of the file containing the desired full-text index. The header of the file contains information about the particular indexing structure, either the BWT, SA-Ψ or STBWT. The indexing structure is encrypted with AES-128, using a key derived with the Password-based Key Derivation Function 2 (PBKDF2) [22], so that the client can enter a password instead of a binary cryptographic key. PBKDF2 repeatedly applies a hash function with a *salt* for an agreed number of iterations. We choose the GCM mode fo operation in order to verify the authenticity of the file. The simple file format we use is reported in Table 4.4. In our implementation, we allow a client to retrieve only the number of occurrences: indeed, the client can freely choose how many positions of such occurrences must be returned. Therefore, the client can choose if the main indexing structure, that allows to retrieve the range of the suffix array that represent a match for the pattern, should be coupled with the suffix array itself, allowing to retrieve the positions of the occurrences, or not, thus retrieving only the number of matches. When the suffix array is included, a value ≥ 4 is employed in the algorithm field (look at Table 4.4). We only encrypt the suffix array (if present), the $\mathcal{C}$ array and the index. The rest of the content is only authenticated since the information it contains is public and part of the leakage model we assume.

Using the `/load` method, the client specifies both the file name and the password. The latter is encrypted by the client and decrypted by the enclave with the symmetric key SK established during the remote attestation procedure. Since SK is ephemeral, this prevents replay attacks that aim at decrypting an index without the consent of the client.

Once the indexing structure is properly loaded into the ORAMs, the client can finally issue queries. The API call `/substring` submits an encrypted query to the server, that returns the initial and final indices of the suffix array that refer to matches of the pattern. If we call them s and e respectively, the client can determine the number of matches evaluating $e - s + 1$, since the range $[s, e]$ is inclusive (see Algorithm 2.3.5). If either $s = e = -1$ or $s > e$, the pattern is not found in the original text. The client can pad the query with as many dummy characters as she wishes to conceal the length of the query. After calling `/substring`, the client can invoke `/suffix` to retrieve the corresponding entries of the suffix array. The suffix array is divided into windows of R entries: the client starts from fetching the window $\lfloor s/R \rfloor$. It can then fetch as many portions of the suffix array as she wishes to conceal the number of matching patterns.

A session can be torn down via the command `close`, that deallocates the ORAMs and destroys the attestation context, preventing a SK from being reused.

The client is implemented in Python and uses the Requests library in order to perform all the API calls. We implemented a small command line interface (CLI) in order to test and benchmark our work.

Chapter 5

Experimental results

The experimental evaluation of our work aims at finding the optimal combination of ORAM primitives and substring search algorithms, as well as assessing the estimated asymptotic complexities and practical performance metrics of our three proposed solutions. In Subsection 5.1 we provide general information about the platform where we perform our tests and the datasets we choose in order to benchmark ObSQRE. Since we implement three different ORAM protocols, in Subsection 5.2 we provide sound values for their free parameters, evaluate their performance in the scenario of enclaved execution and assess the overhead due to doubly obliviousness. In Subsection 5.3 we tune the paramters for the BWT based substring search algorithm, in particular the sample period R and the size of the recursive position map C. Finally, in Subsection 5.4, we directly compare the three different full-text indices we develop.

5.1 Experimental setup

We perform our experiments on an machine endowed with Intel SGX, which means that the performance data showed in this chapter refers to actual execution times on real hardware, instead of being measured on Intel SGX simulator. Even though Intel SGX is included in all the CPUs since the *Skylake* microarchitecture, it requires support from the motherboard firmware to allocate the PRM, that is where enclaves reside. The motherboard of the server where we perform our test supports Intel SGX technology; the machine is equipped with an Intel Xeon E3-1220 v6 CPU at

Table 5.1: Choice of parameters for ORAM primitives. Z is the number of blocks in each bucket, S is the stash size, A is the eviction period and D is the number of dummy blocks in a bucket.

ORAM	**Z**	**S**	**A**	**D**
Path	4	64	–	–
Circuit	3	8	–	–
Ring	4	32	3	6
	8	41	8	13

3GHz, 32GB of RAM and runs Ubuntu 16.04 LTS. We use Intel SGX SDK and PSW v2.5, compiled with Intel's optimized libraries.

In order to narrow down the choice of ORAMs, we perform synthetic benchmarks to establish the time needed by the Access procedure of each implementation. Path ORAM has only two parameters: the number of blocks in a bucket, Z, and the size of the stash S. We fix $Z = 4$ since it is the minimum value of Z that guarantees exponentially decreasing stash overflow probability [36]. The upper bound for stash overflow probability is 1.6^{-S}: by choosing $S = 64$, we achieve a probability of 2^{-43}. For Circuit ORAM, the value $Z = 4$ exhibits a stash size that does not exceed 5 [40]. Following the hints of [40], we opt for a more aggressive combination, picking $Z = 3$ and $S = 8$. We validate these values empirically, experiencing no stash overflows in several runs that perform $\sim 2^{30}$ accesses adopting a round-robin schedule over the block ids: in fact, this access pattern is provably the one that maximizes stash occupancy for all of the ORAM schemes we consider [36]. For Ring ORAM we test two different configurations, deriving the right parameters from Table 2.2 by hinging upon the theoretical overflow analysis done by Ring ORAM authors. First, we choose $Z = 4$ as it directly compares with the competitors. Since this value of Z allows a maximum eviction period of $A = 3$, the second configuration is more aggressive: indeed, we choose $Z = 8$ to get an eviction period of 8, thus assessing how a greater value for A impacts performance in a practical scenario. The configurations employed for all the ORAMs being evaluated are summarised in Table 5.1.

In our tests we also include singly oblivious implementations of Path and Ring ORAMs in order to measure the impact of doubly obliviousness on their performance. We are not planning to exploit them in the final substring search protocol: indeed, the clients of singly oblivious ORAMs are still executed within an enclave, and exposed to side channel leakage that may compromise their security. We keep some optimizations that we introduced for doubly oblivious primitives, for example the *in-place eviction* of the currently fetched path, as well as some other ones that rely on the fact that client resides on the same machine as the server, like the removal of the *XOR trick* for Ring ORAM. The major difference with respect to their doubly oblivious counterpart is that they don't perform linear scans with oblivious swaps, but rather manage the fetched path as a simple array of buckets and the stash as a list of blocks whose size changes dinamically. Hence, by evaluates their performance, we can quantify the overhead due to the implementation of an oblivious client. Since the modifications done to make Circuit ORAM are minimal and straightforward, we are not interested in assessing the overhead introduced by these simple changes, hence focusing only on its doubly oblivious version. For singly oblivious Path and Ring ORAM we adopt the same configurations of Table 5.1, since they are not affected by the way the client is implemented.

We test our substring search algorithms with standard benchmarks of varying length and alphabet size. We consider datasets with different alphabets as we are willing to analyze the impact of alphabet size on the performance of our substring search algorithms. The genomes of *escherichia coli* (`ecoli` in short) [13] and *saccharomyces cerevisiae* (`sacc` in short) [14] only contain the four nucleotides composing the DNA, while the human chromosome 21 (`CHR21` in short) [13] is expressed in FASTA format [8], that include additional characters to account for uncertainty in the genome sequencing process. This chromosome exhibits only a subset of FASTA symbols (namely, 7), therefore the alphabet is only slightly larger than `ecoli` and `sacc`. The Swiss-Prot database (`prot` in short) [38] contains many human proteins, which are encoded using the 25 symbols of the slighty larger protein alphabet. Furthermore it is the only dataset made of several strings, that are concatenated and separated by a symbol that doesn't occur elsewhere in the text (i.e., the end-of-string delimiter symbol). Finally, we choose the corpus of the works of William Shakespeare (`shake` in short) [32] for our tests on natural language. `shake` contains between 89 and 91 characters according to the portion of the text that is considered, but given its high alphabet size, this difference is negligible. Table 5.2 summarizes the chosen

Table 5.2: Substring search datasets

Dataset	**Full name**	**Length (MB)**	**Alphabet**	
			Type	**Size**
`ecoli`	*Escherichia coli*	5.5	Nucleic acid	4
`sacc`	*Saccharomyces cerevisiae*	12.2	Nucleic acid	4
`CHR21`	Human chromosome 21	46.7	FASTA	7
`prot`	Swiss-Prot DB	201.9	Proteins	25
`shake`	Shakespeare corpus	5.5	ASCII	89-91

datasets. We perform the tests over incremental portions of our datasets, ranging from 1MB to 32MB. We don't consider texts of greater size since Ring ORAM with $Z = 8$ has a huge memory footprint. In fact, each bucket stores $Z + D = 21$ records and there is approximately one bucket for every element of a text, hence the solutions based on Ring ORAM require 21x the memory of the initial indexing structures. Given the limited resources of our machine, we opt for smaller datasets in order to provide a fair comparison between all the ORAMs we test, without running out of memory. This is not an issue since in this stage we are mainly interested in analyzing the trends of the different combinations of ORAMs and substring search algorithms: we will show some preliminary tests over bigger datasets after we identifying the best solution among the proposed ones.

5.2 ORAM benchmarking

The benchmarks of the ORAM primitives are taken outside the enclave in order to totally delete the noise generated by the context switch to enclave mode. We generate a dataset of increasing size consisting of 8 byte random values. Since the size of the blocks B is fixed, we consider it a constant and delete it from the complexity bounds of the ORAMs. For every dataset size N, we fully initialize the ORAMs inserting all the elements, in order to reach a steady-state, and we compute the average access time over 1024 accesses. The result of the test is depicted in Figure 5.1.

The linear scan approach was included to show how quickly tree based ORAMs outperform a trivial strategy. The exponential trend is due to the fact that the x-axis, which represents the number of elements N, has a logarithmic scale and a linear scan has complexity $\mathcal{O}(N)$.

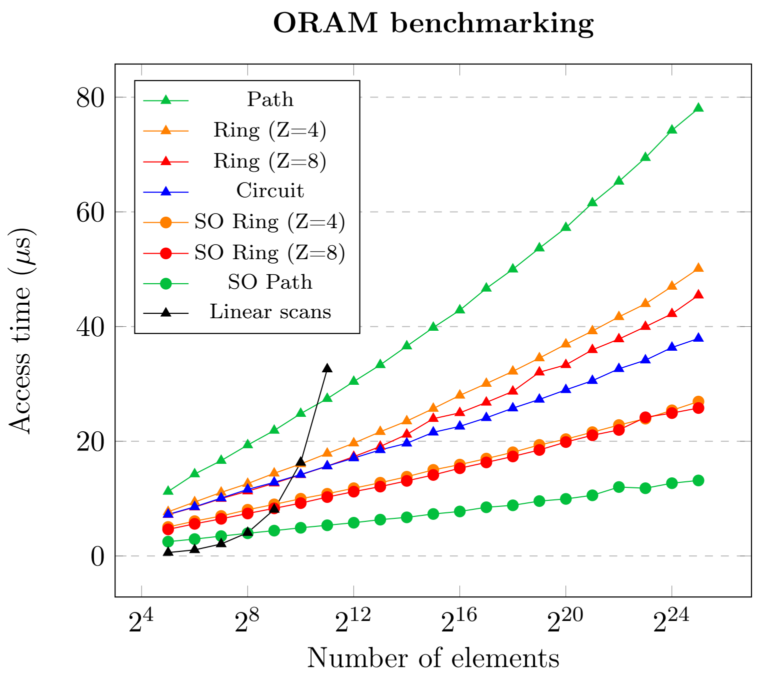

Figure 5.1: Comparison of ORAM primitives. SO stands for singly oblivious versions

As discussed in Section 3.2, eviction is the heaviest operation in terms of computational complexity for doubly oblivious ORAMs. This is confirmed by the experimental results showed in Figure 5.1, as the difference between the performances of the various doubly oblivious ORAMs can be explained by analyzing the computational cost of their eviction procedures. The eviction complexity of doubly oblivious Path ORAM, $\mathcal{O}(Z^2 \cdot \log^2 N + S \cdot Z \cdot \log(N))$, is the highest among all the doubly oblivious primitives, and it's not surprising that it is the slowest. Ring ORAM improves its performance thanks to the fact that evictions are performed every A accesses. Moreover, our optimizations allow to save multiplicative factors of Z and $\log N$, yielding a complexity of $\mathcal{O}(Z \cdot \log^2 N + S \cdot \log(N) + Z \cdot (Z + D) \cdot \log N)$ for a single eviction. Even if Ring ORAM with $Z = 4$ performs more than twice the evictions of $Z = 8$, the latter also exhibits a much greater value for D, that impacts on eviction complexity. Thus, the performance improvement of Ring ORAM with $Z = 8$ over Ring ORAM with $Z = 4$ is not so significant. Circuit ORAM has the best asymptotic bounds, hence it prevails for larger N. Since its complexity is logarithmic in N, it is the only primitive that does not exhibit a slight quadratic trend among doubly oblivious ones. Ring ORAM with $Z = 8$ is slightly better for $N \leq 2^{11}$, though.

Unsurprisingly, singly oblivious primitives are much faster. Despite its longer eviction period A, Ring ORAM performs worse than Path ORAM, that in fact is a much simpler protocol. The reason is that in our local scenario bandwidth costs have not a high impact on performance, since they are just memory to memory transfers. Ring ORAM, on the other hand, improves on a remote scenario in which network latency absorbs the vast majority of execution time. Hence, the simpler control flow of Path ORAM results in shorter execution times. Figure 5.2 shows the slowdown of the doubly oblivious primitives with respect to their singly oblivious counterparts. Path ORAM is the fastest singly oblivious ORAM due to its simple structure, but also the slowest doubly oblivious since its eviction has a cost that is roughly $\mathcal{O}(\log^2(N))$ and is performed after each Access. Hence, its overhead nearly reaches a factor of 6 and is doomed to grow for larger ORAM sizes, since the Access complexity of a singly oblivious Path ORAM is $\mathcal{O}(\log(N))$, resulting in a ratio that will be $\mathcal{O}(\log(N))$ as well. Indeed, the overhead resembles a straight line in the plot since the x-axis is in logarithmic scale. While the overhead of Ring ORAM is a straight line for the same reasons outlined for Path ORAM, the resulting slopes are much gentler thanks to several optimizations that reduce the multiplicative factor hidden in the asymptotic complexity of doubly oblivious evictions. Specifically, as evictions are the operations where there is more gap between the single and doubly oblivious Ring ORAMs, amortizing them over A accesses has a greater impact on the doubly oblivious ORAMs. Indeed, evictions are relatively cheap operations in a singly oblivious implementation, hence adopting a higher A produces negligible effects in our local setting; conversely, a longer eviction period yields more benefits in the doubly oblivious version. This motivation is confirmed by observing the slowdown for the two configurations we consider in our tests for Ring ORAM: despite the slowdown of the doubly oblivious versions is similar for both configurations, Ring ORAM with $Z = 8$ exhibits a slightly smaller gap due to its longer eviction period.

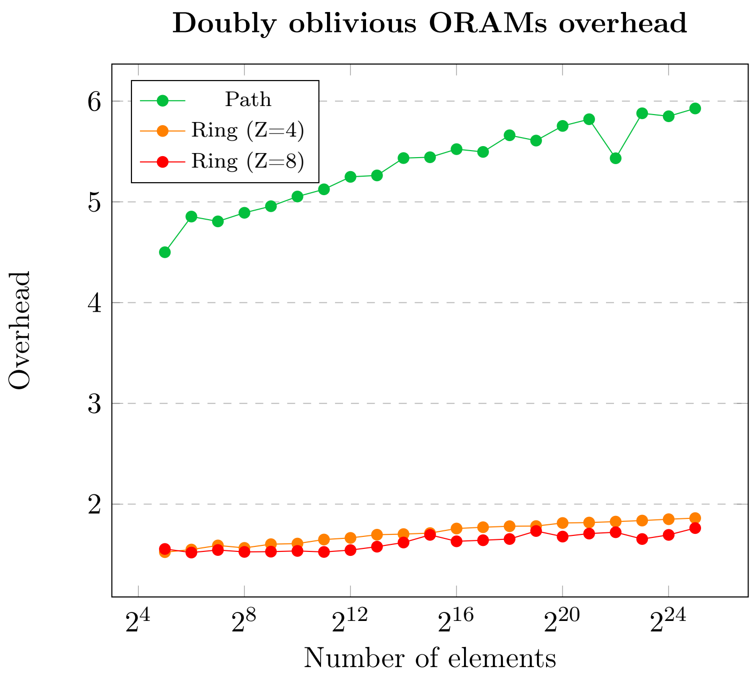

Figure 5.2: Overhead of doubly oblivious ORAMs with respect to their singly oblivious counterpart

5.3 BWT parameter tuning

The complexity of the BWT based algorithm depends on the parameter R, i.e. the sampling period of the matrix M that stores the ranks for backwards search. R needs to be carefully chosen since it has several side effects that may influence the performance of this solution, namely:

1. bigger values for R lead to a bigger block size of the ORAM, since each block will contain R consecutive characters from the BWT of the text;
2. bigger values for R increase the number of characters of the BWT which need to be scanned linearly during the rank computation, as all the R characters in a block must always be inspected;
3. bigger values for R decrease the number of blocks of the ORAM that is $N = \frac{|T|}{R}$, for a text T.

In short, the block size and the number of characters to be scanned during the rank increase with R and decay the performance, but on the other hand the number of required ORAM blocks decreases for bigger values R, determining a better access time. It is not possible to forecast where the correct trade-off lies, since the constant factor hidden behind these operations is unpredictable.

Furthermore, BWT based backward search employs a recursive position map, as described in Subsection 2.2.4. The number of elements stored in each chunk of the recursive position map, C, is another free parameter which needs to be aptly tuned. We observe that these two parameters, R and C, are intertwined: we expect that the ideal value for C depends on the number of blocks stored in the ORAM, which in turn depends on R, as outlined in the third point of the listing above. Performing an exhaustive search in both these dimensions would require too much time. Nonetheless, these two parameters are related only by the number of blocks of the ORAM: hence, we first find the ideal size of C for ORAMs with various number of blocks; then, for every R, we compute the the number of blocks as $N = \frac{|T|}{R}$ and choose the optimal value C for that number of blocks.

5.3.1 Calibration of C

In order to find the optimal value for C, we run synthetic benchmarks for the most promising doubly oblivious primitives, i.e. the Ring ORAMs with the two configurations ($Z = 4$ and $Z = 8$) and the Circuit ORAM. We test ORAMs of varying sizes between 2^9 and 2^{22} elements in order to cover all the combinations of text sizes and possible sample periods in our datasets. For each ORAM and dataset size, we try out all the C values in the range $\{2^2, 2^3, \ldots, 2^{10}\}$. We choose 2^{10} as the upper bound because Figure 5.1 shows that linear scan becomes less efficient than using an ORAM at that point. Since obliviousness requires to scan linearly all the entries in the position map chunks, the optimal value must necessarily be in the tested range.

The block size of the ORAM that stores the data is 8 bytes (as in the previous test): in fact we don't want its access time to influence our results. We fully initialize all the ORAMs, to reach a steady-state, and collect 1024 samples of the access time. In Figure 5.4 we show the results for dataset sizes between 2^{17} and 2^{22}, while the full results are shown in Appendix A.

We observe that the shape of the plots is always similar, with the optimal values always lying in the range $[2^5, 2^7]$. It is possible to notice that Ring ORAM with $Z = 4$ is always slower than the other two solutions, confirming the results shown in Figure 5.1. However, Ring ORAM with $Z = 8$ is slightly faster than Circuit, contradicting the trend of the previous benchmarks. Since recursion employs ORAMs of increasing sizes, specifically the ORAM for the i-th level stores C^i blocks, the first levels are quite small. The better performance of Ring ORAM when the number of blocks is small is indeed confirmed by the benchmarks of the various ORAMs showed in Figure 5.1: here, Ring ORAM performs better than Circuit, up to 2^{10} blocks. Furthermore, we observe that this effect slowly reduces as we approach to 2^{22}, again suggesting the better scalability of Circuit ORAM against Ring ORAM. Indeed, we observe that the performance of Ring and Circuit ORAMs for the optimal values of C become closer when the dataset size increases. In order to prove this empirical evidence, we perform further tests increasing the number of elements in the last level ORAM to 2^{25}. The results are shown in Figure 5.3. We performed the tests only for Circuit ORAM and Ring ORAM with $Z = 8$, for a reduced range of C between 2^4 and 2^8, as all the optimal values found in previous tests were included in this range. The results show that the access time of Circuit ORAM for the optimal value $C = 32$ is lower than Ring ORAM. Since the asymptotic bounds of Circuit ORAM are better, we expect this gap to grow in its favor when the number of blocks increases.

We extract the optimal values from all the plots and report them in Table 5.3. Different optimal values (2^9 and 2^{10}) are found for datasets with 2^9 and 2^{10} blocks, respectively: this effect is due to the fact that the position map is totally stored on the client side, and no intermediate ORAMs are accessed. It's easy to see that the optimal value for C doesn't change if we consider different ORAM primitives, meaning that the balance between the number linear scans of recursive position map, the number of levels and their size remains the same.

5.3.2 Calibration of R

The tests to find the the optimal sample period R are performed on all the real datasets employed in our evaluation: the results reported here are actual query times taken from the BWT based algorithm (see Section 3.3.1)

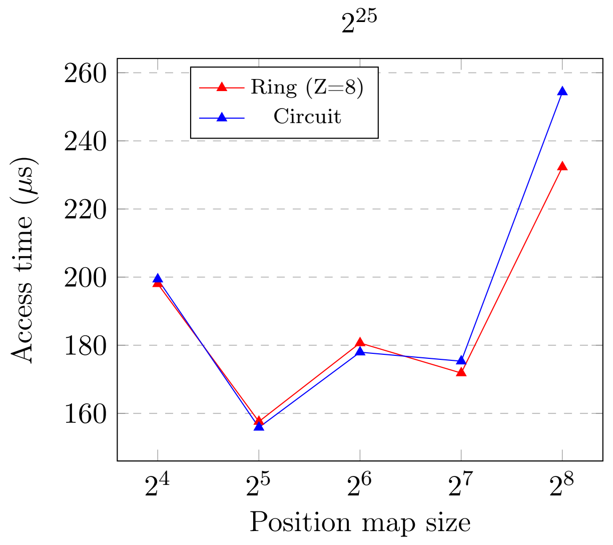

Figure 5.3: Comparison of recursive position map for Ring and Circuit ORAM with 2^{25} elements

Table 5.3: Optimal values for C

Size	**Circuit**		**Ring (Z=4)**		**Ring (Z=8)**	
	$\log_2 C$	**time (μs)**	$\log_2 C$	**time (μs)**	$\log_2 C$	**time (μs)**
2^9	9	14.367	9	15.452	9	13.838
2^{10}	10	16.746	10	18.308	10	16.337
2^{11}	6	31.040	6	32.230	6	29.288
2^{12}	6	35	6	36.915	6	33.478
2^{13}	7	44.435	7	45.865	7	42.352
2^{14}	7	50.263	7	52.386	7	48.470
2^{15}	5	55.425	5	60.159	5	54.146
2^{16}	6	63.877	6	68.882	6	62.187
2^{17}	6	69.730	6	76.598	6	69.191
2^{18}	6	77.834	6	85.185	6	77.183
2^{19}	5	91.742	5	99.446	5	89.659
2^{20}	5	98.763	5	108.283	5	97.459
2^{21}	6	116.413	6	125.978	6	113.606
2^{22}	6	123.677	6	133.299	6	122.182

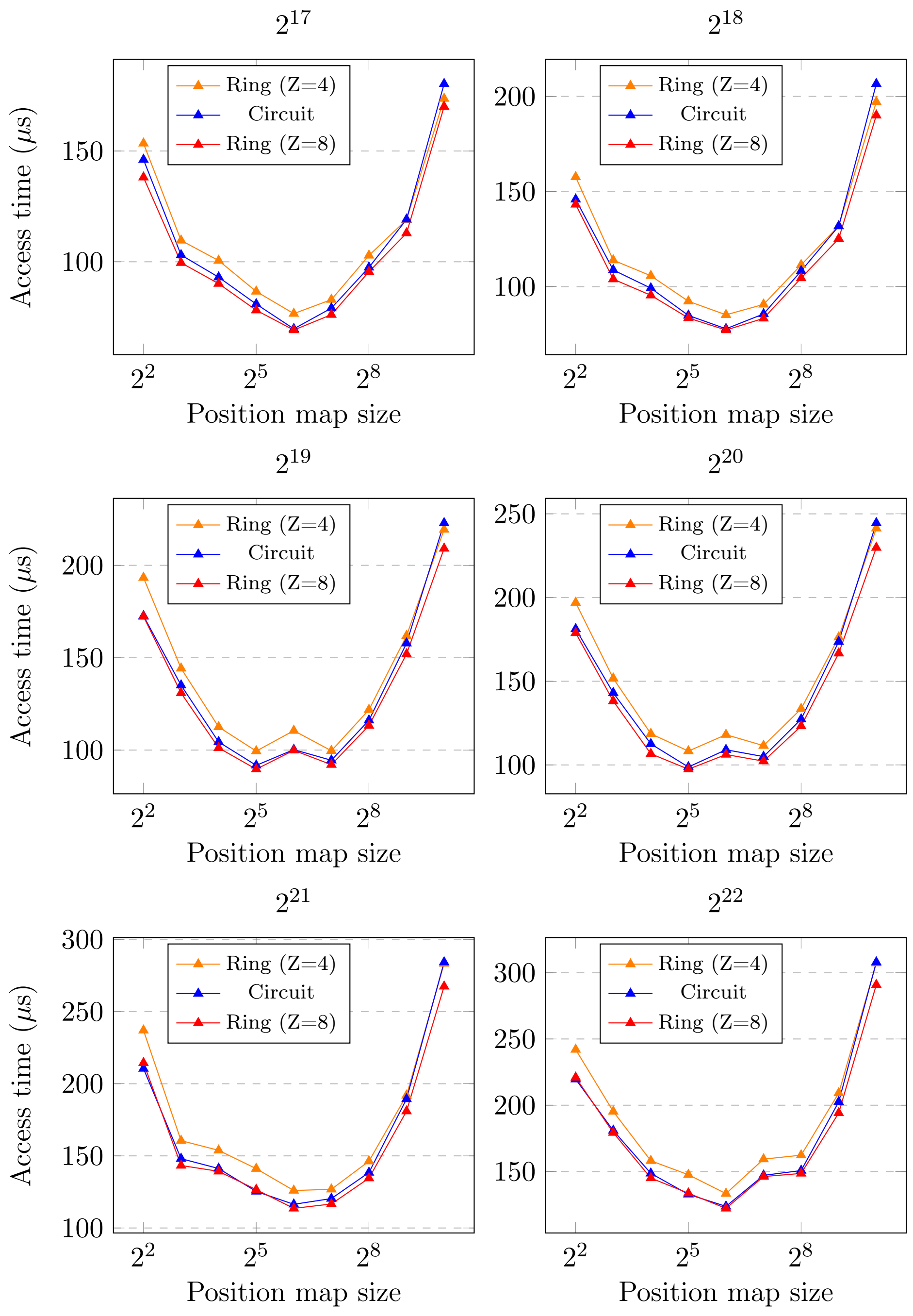

Figure 5.4: Comparison of recursive position map for ORAMs with the number of blocks ranging from 2^{17} to 2^{22}

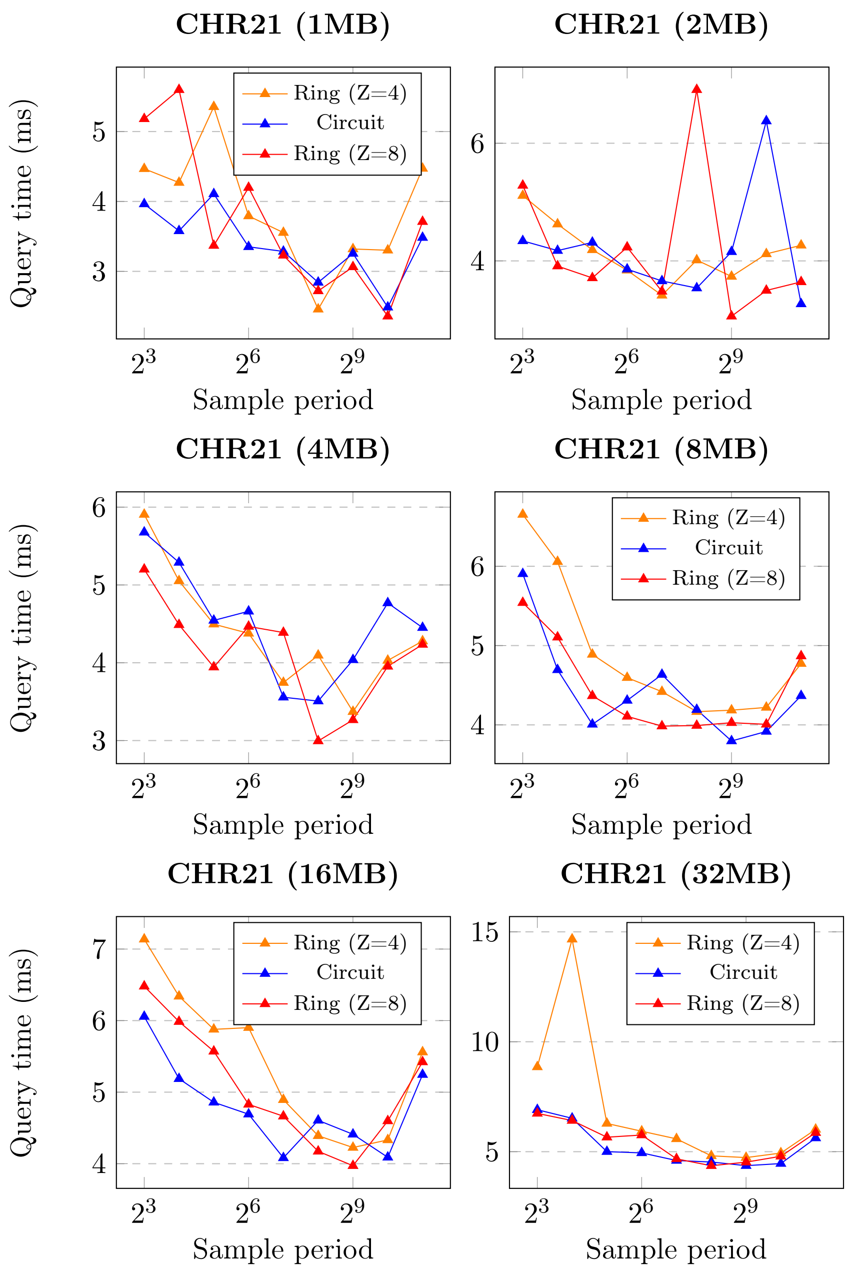

Figure 5.5: Calibration of the sampling period R for `CHR21`

running on Intel SGX, averaged over 32 queries of 24 characters to remove noise in the measurements. We consider portions of the original texts of increasing size to analyze the impact on the optimal value of R. We consider values of R in the interval $\{2^3, 2^4, \ldots, 2^{10}\}$. We pick 8 as lowest value of R since it is comparable to the size of the smallest alphabet of our testcases, which is 4 for `ecoli` and `sacc`, that only contain nucleotides. This choice is motivated by the fact that all the substring search algorithms we consider already perform a scan over the $|\Sigma|$ entries of array $\mathcal{C}$ for each query character. Hence, a smaller value of R would not be so relevant, since the complexity of linear scans would already be dominated by the size of the alphabet. Figures 5.5, 5.6, 5.7 show the results of this analysis for `CHR21`, `prot` and `shake`, respectively, while the plots for `ecoli` and `sacc` are reported in Appendix A. In general, these benchmarks appear more noisy and irregular than the ones analyzed so far. One hypothesis which may explain this behavior is that the execution inside secure enclaves introduces some non deterministic effects, due to caching effects of memory pages of ORAM blocks, which may avoid costly page table walks.

It is possible to observe that larger alphabets prevent R from achieving better performance as it grows. Especially for moderate to high text sizes, both the `prot` and `shake` (Figure 5.7) datasets exhibit a regularly growing query time for increasing R. We ascribe this behavior to the fact that a single character requires more bits to be encoded, specifically $\log_2 |\Sigma|$, which makes the overhead due to the higher block size in the ORAM overwhelming with respect to the benefits on the access time given by a smaller number of blocks.

On the other hand, datasets over smaller alphabets, such as the genomic ones (Figure 5.5), still benefit from increasing sampling periods, due to the fact that their block size doesn't grow too much. In fact, an increasing R allows to allocate less blocks, thus optimizing the access time to ORAMs. This is mostly true for Ring ORAM, which exhibits good performance when containing a small number of blocks: in this case, the additional time spent in longer linear scans is less than the one saved in the memory access. Circuit ORAM, on the other hand, scales better with increasing number of blocks, and suffers from longer linear scans for texts whose size is at most 4 MB. However, all these observations exhibits outliers, that complicate the

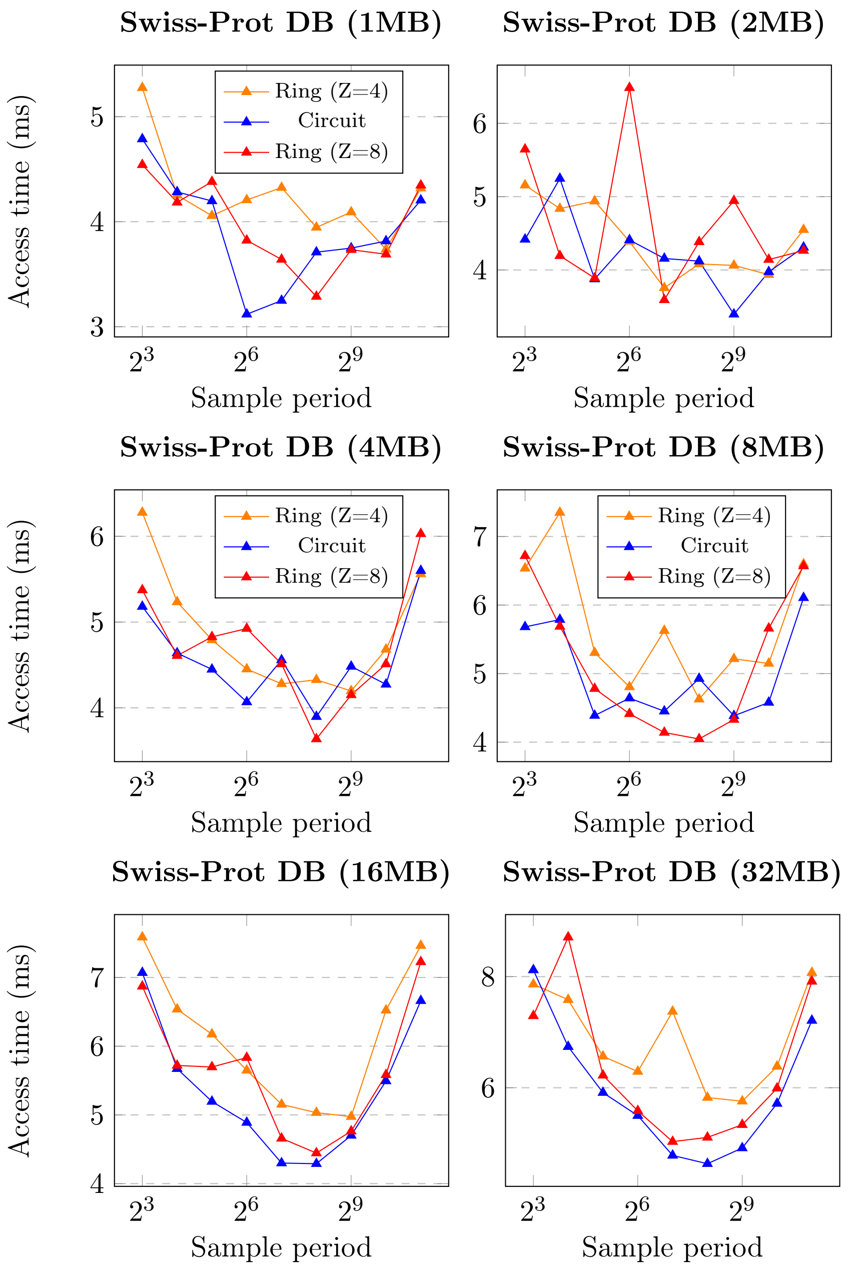

Figure 5.6: Calibration of the sampling period R for `prot`

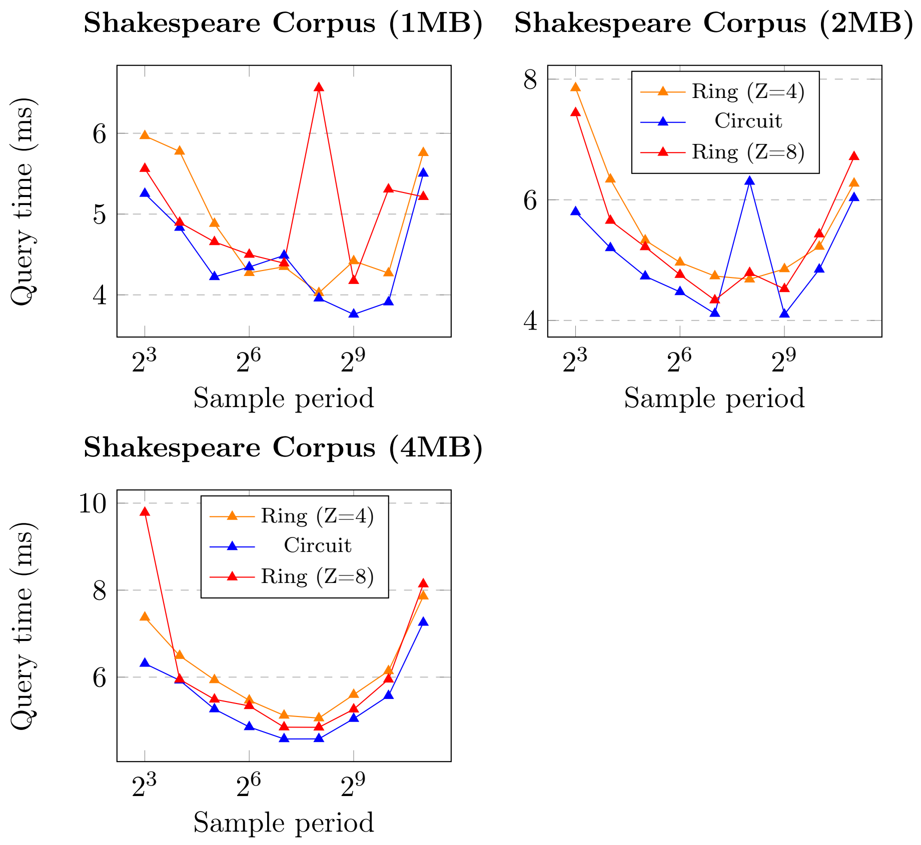

Figure 5.7: Calibration of the sampling period R for `shake`

analysis.

Circuit ORAM usually performs better for optimal choices of parameters, and notably for larger texts and alphabets, which prevent the usage of a high R. Whenever there is a moderate text size and large R, Ring ORAM with $Z = 8$ competes on equal terms. On the other hand, Ring ORAM with $Z = 4$ is slower and less stable, since it exhibits the highest spikes. Overall, it is hard to devise a general rule to establish which alternative is the best, since there are three different parameters that depend on R. The best choice of parameters, that leverage different properties for each type of ORAMs, achieve the same performance levels for the data size we consider.

Table 5.4 summarizes the values of R that minimize the query time for each dataset and ORAM. The optimal size of the recursive position map is derived from Table 5.3: given a text T and a sample period R, the number of elements in the last level ORAM will be $N = \frac{|T|}{R}$. For each ORAM protocol,

Table 5.4: Optimal combinations of R and C for each dataset

Dataset	**Size (MB)**	**Circuit**		**Ring (Z=8)**		**Ring (Z=4)**	
		R	$\log_2 C$	R	$\log_2 C$	R	$\log_2 C$
`ecoli`	1	1024	10	1024	10	512	6
	2	64	5	1024	6	256	7
	4	128	5	64	6	256	7
`sacc`	1	512	6	512	6	1024	10
	2	512	6	1024	6	128	7
	4	64	6	512	7	256	7
	8	512	7	1024	7	1024	7
`CHR21`	1	1024	10	1024	10	256	6
	2	2048	10	512	6	128	6
	4	256	7	256	7	512	7
	8	512	7	128	6	256	5
	16	128	6	512	5	512	5
	32	512	6	256	6	512	6
`prot`	1	64	7	256	6	1024	10
	2	512	6	128	6	128	6
	4	256	7	256	7	512	7
	8	512	7	256	5	256	5
	16	256	6	256	6	512	5
	32	256	6	128	6	512	6
`shake`	1	512	6	512	6	256	6
	2	512	6	128	7	256	7
	4	128	5	256	7	256	7

we choose the value of C based on this N.

5.4 Substring search algorithms

The comparison among the substring search algorithms takes place on the final application running on Intel SGX. The execution times are the average of 32 queries for the same pattern. In our tests, we always employ patterns of the same length: indeed, since all the three substring search algorithms perform a number of iterations which is equal to the length of the pattern, it is straightforward and not particularly interesting to show the linear dependence between the length of the pattern and the query execution time. Specifically, the patterns used for the different datasets are 24 characters long, padded with dummies when necessary. While the dummies contribute with the same amount of computation of a real characters, they obviously do not have any side effect on the results of the query. We choose this query length since its size allows to construct substrings with a reasonable number of occurrences for all the given datasets without resorting too much to padding. Searching for patterns which occur in the texts provides a fast way to verify the correctness of our algorithms.

In Figure 5.8, 5.9 and 5.10 we plot the results for the dataset `CHR21`, `prot` and `shake` respectively. The ones of `ecoli` and `sacc` can be found in Appendix A since they are akin to the ones of `CHR21`. It is evident that the BWT based algorithm outperforms all the other ones for all our test cases, achieving a speedup between 3x and 5x when compared to the best ODS based algorithm. Even if its cost per query character is $\mathcal{O}(\log^2(N))$, hence similar to the one of ODS's based algorithms, it has much better constant factors in front of it. Indeed, the joint optimization of C and R allows to perform fine tuning of this solution: once that the value of R that balances the cost of linear scans with the number blocks of the ORAM is found, it is possible to choose the C that optimizes the retrieval of the right leaf id from the levels of the recursion. The optimization of these important parameters significantly contribute to the approximately $6\times$ speed-up observed in the results. The performance levels guaranteed by the different ORAM primitives are comparable, as observed in Subsection 5.3.2.

On the other hand, ODS based approach exhibit a regular trend. For SA-Ψ, Circuit ORAM is always faster that the Ring ORAMs, and for the latter, $Z = 8$ yields better results than $Z = 4$. The same holds for STBWT. These two algorithms have the same asymptotic complexity, but the shape of their ODS is slightly different: in fact, SA-Ψ consists of a complete binary tree containing N elements, while STBWT arranges the occurrences of each character in a separate complete binary tree. The first structure is deeper, while the split trees are shallower, but have bulkier levels. The former approach seems to be the better one due to a lower access time, but we also identify another reason for its performance. Backwards search based on SA-Ψ performs a linear scan over $|\Sigma|$ elements to fetch an entry from the array $\mathcal{C}$. However, STBWT requires to additionally scan the array containing the roots of each of the $|\Sigma|$ search trees that compose the ODS. Hence, it

Table 5.5: Comparison of query time for alphabet size

Dataset	**Alphabet size**	**BWT (ms)**	**SA-Ψ (ms)**	**STBWT (ms)**
`sacc`	4	3.31	16.13	17.67
`ecoli`	4	3.19	16.19	17.27
`CHR21`	7	2.99	16.04	18.40
`prot`	25	3.63	16.22	18.98
`shake`	91	4.57	16.17	23.27

performs twice the linear scans of SA-Ψ. This effect is visible if we observe the benchmarks of `shake`: in this case, the slowest ORAM of SA-Ψ is faster than any STBWT solution. On the other hand, by reducing progressively the alphabet size, we notice how the fastest STBWT implementation beats the slowest SA-Ψ ORAM (for `prot` and `CHR21`). This observation highlights how linear scans have an impact on performance, along with alphabet size.

Since all the patterns we search have the same length, we may better assess how performance changes according to alphabet size. In order to perform the comparison, we pick the best query times for each algorithm for text sizes of 4 MB, that is common to all the considered datasets. The results are shown in Table 5.5. While the STBWT is the solution that exhibits the highest sensitivity to alphabet size, because of the previous observation, SA-Ψ which is less affected by the alphabet size: we ascribe this result to the fact that it performs only one linear scan over an array with $|\Sigma|$ elements per each iteration of the backward search, namely the linear scan on the array $\mathcal{C}$ (Algorithm 3.3.5). Conversely, the BWT requires two additional linear scans of arrays with $|\Sigma|$ elements in each iteration of the backward search algorithm (Algorithm 2.3.5): indeed, each $rank_{BWT,\sigma}(x)$ query needs to linearly scan the array $M_R[\lfloor \frac{x}{R} \rfloor].count$, which has $|\Sigma|$ entries. However, the alphabet size influences also the choice of the sample period R, which has also an impact on the performance of this algorithm: specifically, R tends to be greater than the alphabet size is small, thus the cost of linear scans of $|\Sigma|$ elements may be superseded by the R characters which need to be linearly inspected when fetching an entry from M_R. Therefore, the choice of an optimal sample period may reduce the impact of the alphabet size on the performance of the queries. Ultimately, we can conclude that even if negligible, the alphabet size has an influence in the query time of the algorithms.

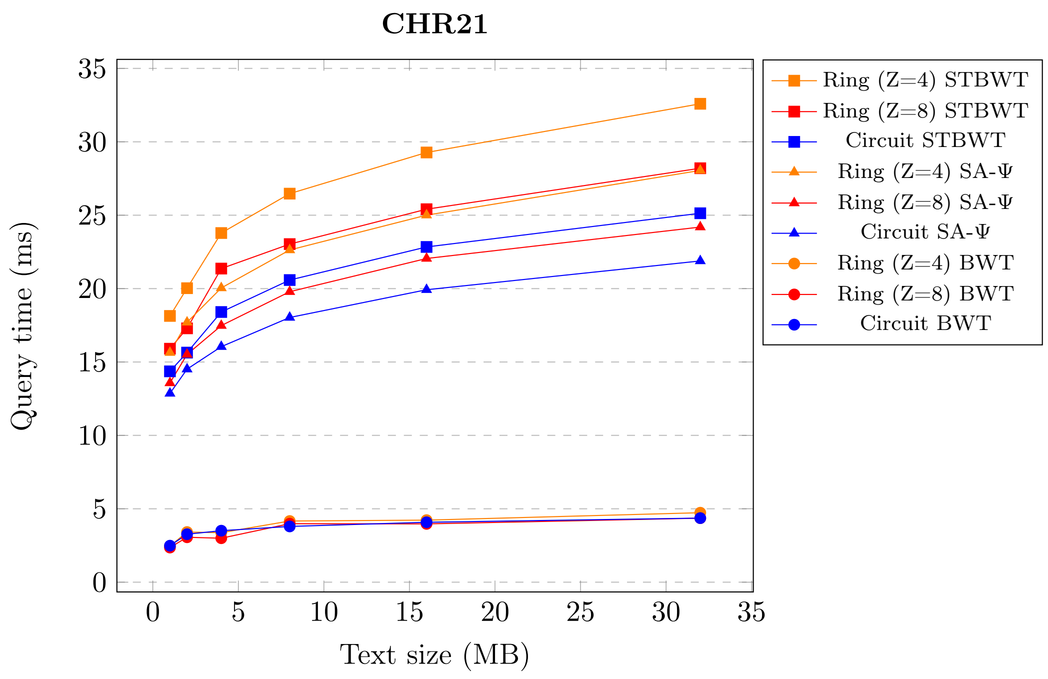

Figure 5.8: Comparison of ObSQRE text indices for `CHR21`

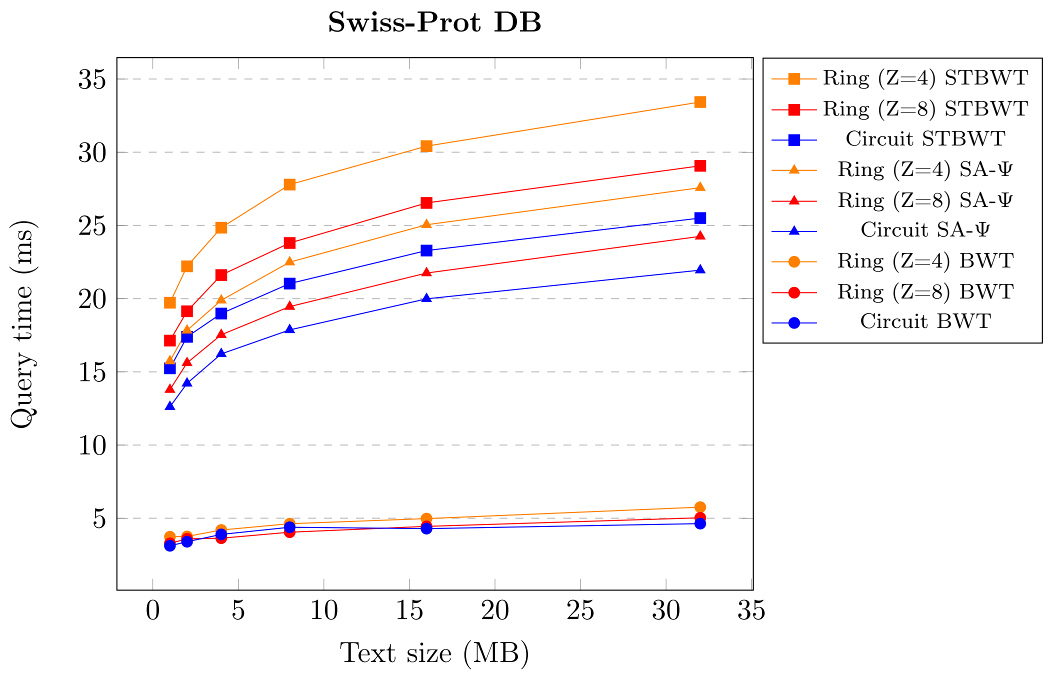

Figure 5.9: Comparison of ObSQRE text indices for `prot`

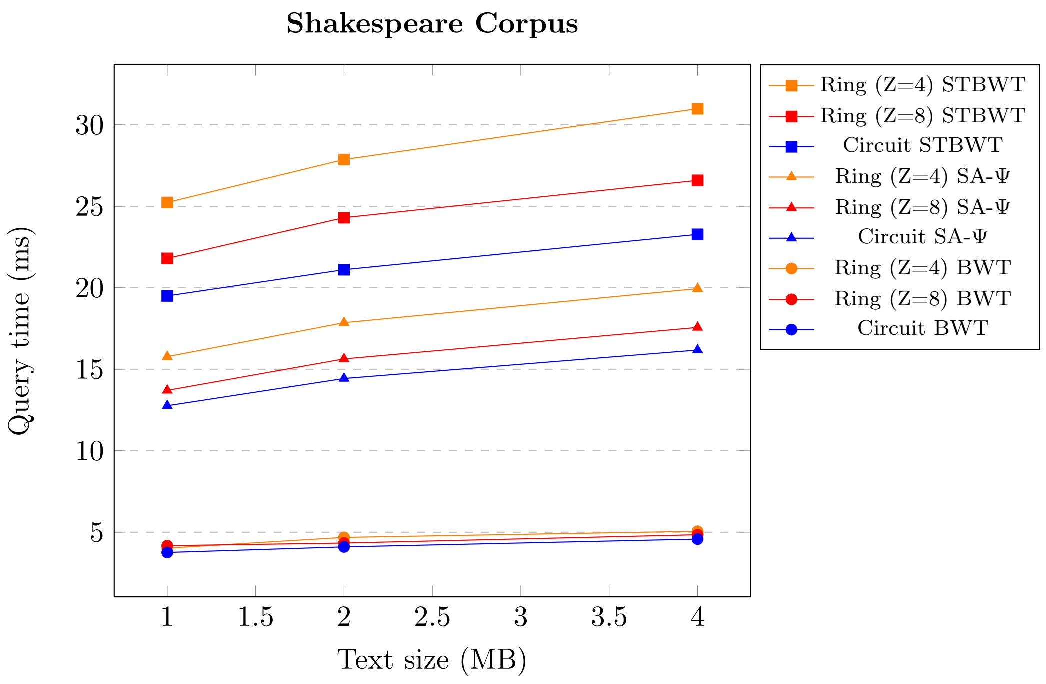

Figure 5.10: Comparison of ObSQRE text indices for `shake`

We provide comparison of our oblivious text indices with a baseline implementation with no security guarantees, which searches the same pattern on the same unencrypted text index, without hinging upon ORAM or Intel SGX. We recall that given a pattern of length m and a text of length n, the complexity of the BWT based algorithm is $\mathcal{O}(m)$, while both SA-Ψ and STBWT run in $\mathcal{O}(m \cdot \log(n))$. The results obtained for `CHR21` and `shake` are depicted in Figures 5.11 and 5.12 respectively, while the rest of the plots are reported in Appendix A. Since the unprotected version of the BWT based approach saves a multiplicative factor of $\log^2(n)$, it is the one which exhibits the largest overhead, of about 7000$\times$ for `CHR21` and 10000x for `shake`. We observe that the maximum size of the latter is only 4 MB, yet its overhead is much greater than `CHR21`, which reaches 32 MB. This is due to the alphabet sizes of the two datasets: in fact, `shake` comprises up to 91 distinct ASCII characters, while `CHR21` only a subset of 7 FASTA characters. We argue that the time spent scanning linearly the array $\mathcal{C}$ and the sample $M_R[\lfloor \frac{x}{R} \rfloor].count$ in the oblivious versions indeed produces massive effects on the overall time penalty, since their length depends on the size of the alphabet. In general, we notice that the overhead of each oblivious algorithm increases with the size of the input alphabet due to the longer linear scans, which surprisingly have a strong impact on the performance. We observe that the overhead of STBWT is much greater that the one of SA-Ψ, even though they should incur the same penalty of $\log(n)$. In fact, the unprotected version of STBWT performs the binary search on each of the occ_σ array separately, while in the version implemented within ObSQRE, the ODS that stores the search trees is shared among all the characters of the alphabet in order to conceal which one is being processed. This produces bulkier ORAMs for each level of the ODS, further penalizing the oblivious version. SA-Ψ is the algorithm that exhibits the smallest overhead in all of the datasets. This is partially due to the fact that it is less prone to the size of the alphabet, but also depends on the way it is implemented in the unprotected version. In fact, we adopt Algorithm 3.3.5 instead of Algorithm 3.3.4 even if the former carries useless overhead. This choice is motivated by the need to validate Algorithm 3.3.5, that is fairly exotic in its search criterion, rather than implementing its most optimized baseline version. We argue that if we performed the search over the Ψ array only choosing the entries corresponding to the current character, we would obtain an overhead similar to the one of STBWT.

Ultimately, the performance loss introduced by ORAMs, Intel SGX and obliviousness ranges from about $2000\times$ to $10000\times$, which is considerable. However, we highlight that ObSQRE achieves performance levels that make it usable in real world scenarios, despite its huge overhead.

In this regard, we perform further tests over 32 MB of `CHR21`, searching for a pattern composed of 3050 nucleotides. Each sequence of 3 nucleotides corresponds to a single amino acid, while a human protein is approximately composed of 1000 amino acids. Hence, a pattern of ~ 3000 characters, that encodes a gene within a human chromosome, represents a query of practical interest in the genomic setting. We opted for the BWT based approach combined with Circuit ORAM, since this is the best solution among all the alternatives we provide, and we choose the optimal parametrization according to Table 5.4. We search for the same substring 128 times and average the running times we obtain. ObSQRE executes our queries in 537 ms, thus requiring 176μs for each character. Since Circuit ORAM is the least memory hungry ORAM, we are able to run the search over the larger human

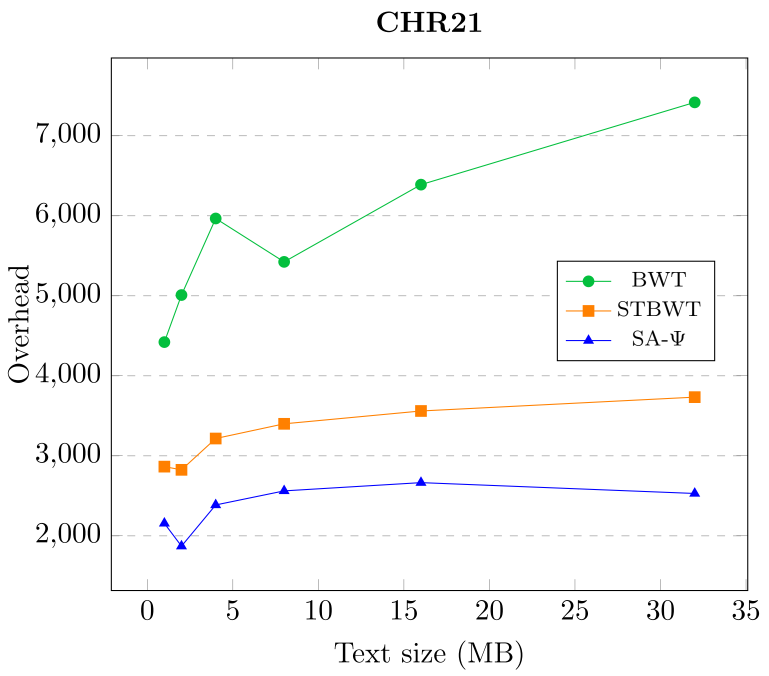

Figure 5.11: Overhead of ObSQRE over baseline for `CHR21`

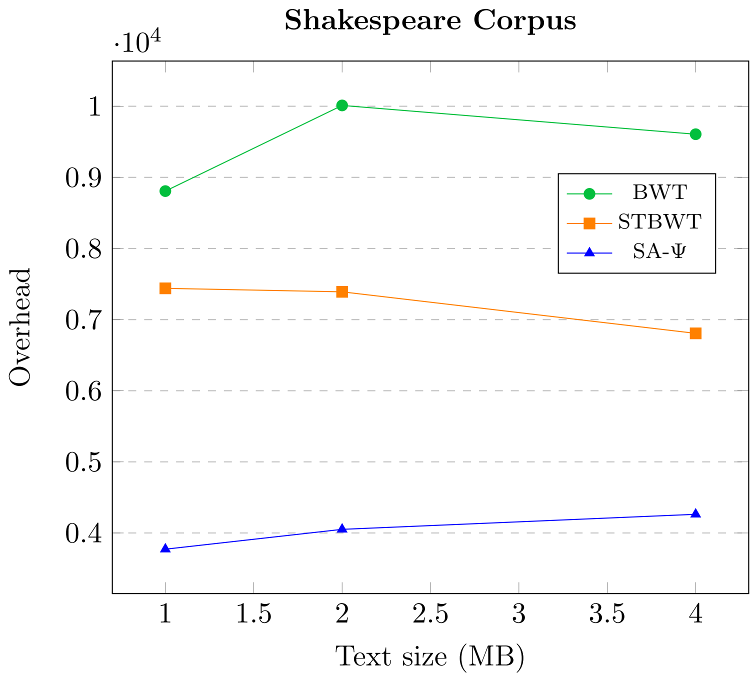

Figure 5.12: Overhead of ObSQRE over baseline for `shake`

chromosome 1, whose size is 249 MB. The average execution time we obtain is 775 ms, i.e. about 254μs per query character. This last test shows that our protocol provides good scalability even when the size of the input text reaches considerable values.

Chapter 6

Conclusion

In this work we present an efficient protocol that solves the problem of obliviously searching for occurrences of a substring in a text that is stored remotely. We address the scenario of large texts that are not be processed locally by a memory constrained client, and thus require outsourcing of confidential data. To provide good perfomance, we opt for substring search algorithms that allow to execute queries in sublinear time with respect to the text length, thanks to indexing structures that are built offline and do not change during the execution of the queries. In order to guarantee data confidentiality, we perform the computations within a trusted execution environment provided by Intel SGX technology: in order to achieve strong confidentiality and integrity guarantees, we directly address the information leakage due to side channels, that allows a root level adversary to observe the memory access pattern in Intel SGX based applications. In this regard, we rely on strong and provably secure cryptographic primitives, namely ORAMs, that allow to totally conceal the memory access pattern of an application, which turns out to be the main source of side channel leakage. The usage of secure enclaves allows to run the ORAM client on the same machine that stores the bulk of the data: this approach allows to overcome the main limitation of ORAMs, i.e. the bandwidth blowup and the need for several network round trips, which may severely impact the performance of the final application. In particular, our protocol allows to retrieve the number of occurrences of a pattern in a single request. On the other hand, the usual ORAM protocols need to be reviewed and modified to take into account that the ORAM client is exposed to side channel leakage as well, potentially compromising the security of these constructions.

We compare several ORAM protocols assessing their theoretical and practical efficiency in the context of secure enclaves. In particular, we implement two existing doubly oblivious ORAMs, namely Path and Circuit ORAM, and present a doubly oblivious version of Ring ORAM, optimizing its construction to take into account our scenario. We discuss how ORAMs can be used to wrap the full-text indices we choose and compare the effectiveness of three different substring search algorithm. While two of them leverage the concept of ODS, the third one resorts to a more traditional recursive position map. Even though all of them achieve the same asymptotic complexity of $\mathcal{O}(m \cdot \log^2 n)$, where m is the length of the pattern and n the size of the text, the experimental evidence highlights that the algorithm based on recursive position map is the most efficient, since it hides much better multiplicative constants. In particular, we experimentally verified that our best solution is able to find the occurrences of a protein sequence (represented by approximately 3000 nucleotide) over 32 MB of genomics data in only 500 milliseconds, which means that a single query character requires a few hundreds μs to be processed. To validate our solution, we perform our tests on datasets of varying length and we analyze the correlation between the achieved performance and the alphabet size. In particular, we find out that alphabet size does not influence significantly the search time, and one of the algorithms in particular, SA-Ψ, does not exhibit any perfomance gap when querying string over a larger alphabet. Our final solution only leaks the length of the text and the size of the alphabet, which are usually assumed to be public parameters, while guaranteeing the confidentiality of the outsourced text, the queried substring and the occurrences of the substring. The length of the substring and the number of occurrences, which could be inferred, respectively, by the number of iterations of the substring search algorithm being employed and by the amount of elements returned to the the client, are partially concealed by adopting proper padding strategies.

Lastly, we carefully reviewed the remote attestation procedure in order to assess its security. Intel primarily designed it for DRM or similar purposes: in the scenario they envision, the client of the application runs an enclave in order to access protected data (e.g. copyrighted works) distributed by a service provider. The service provider, which coincides with the enclave developer or ISV, performs remote attestation before sending the contents to the client, ensuring that she will not be able to abuse it or violate license agreements. In the cloud computing scenario, instead, the tenant (or client) allocates an enclave on a remote server to prevent it from leaking confidential data. However, in general the client is not the same entity as the ISV, and thus she has not access to the IAS, that are extremely important in verifying attestation signatures. We argue that it is infeasible that each client recompiles an enclave and gains access to the IAS, since it requires a certificate issued by a Certification Authority. Hence, we explicitly address this problem, proving that even when the requirements of remote attestation change, the procedure originally devised by Intel is secure even in the cloud server to client scenario. Most of the related works based on Intel SGX either take for granted that the procedure is valid or they do not even consider this remote attestation issue: to the best of our knowledge, we are the first to perform such an analysis.

In the future we intend to explore additional substring search algorithms based on backwards search, exploiting advanced data structures in order to perform rank queries. We also want to perform extensive tests over datasets of increasing size in order to assess the memory footprint and the scalability of the currently available solutions. We plan to extend our work by exploring concurrent ORAMs [6, 30], that unlike the conventional ones we have presented, allow to access several data at the same time. This would provide the possibility to run multiple queries over the same dataset simultaneously, in turn allowing to both open up to high throughput implementations for a single user as well as taking into account multi-user scenarios. Furthermore, we intend to use the ORAM primitives we develop in other application scenarios that can benefit from being executed inside secure enclaves.

Appendix A

Experimental results addendum

Here we report the experimental results that were omitted in Chapter 5 due to space limits. In particular, Figures A.1 and A.2 complete the benchmarks aiming at finding the optimal size of the recursive position map C. Figures A.3 and A.4 represent the calibration of R for the datasets `ecoli` and `sacc`, that follow the same considerations of `CHR21` due to the fact that they all are genomic datasets. For the same reason, the results of ObSQRE for `ecoli` and `sacc` are depicted in Figures A.5 and A.6. Finally, Figures A.7, A.8 and A.9 represent the overhead of ObSQRE against the unprotected baseline for the datasets `ecoli`, `sacc` and `prot`.

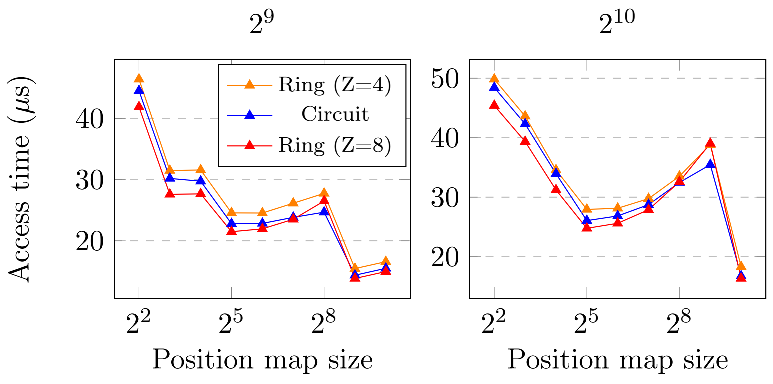

Figure A.1: Comparison of recursive position map sizes for 2^9 and 2^{10}

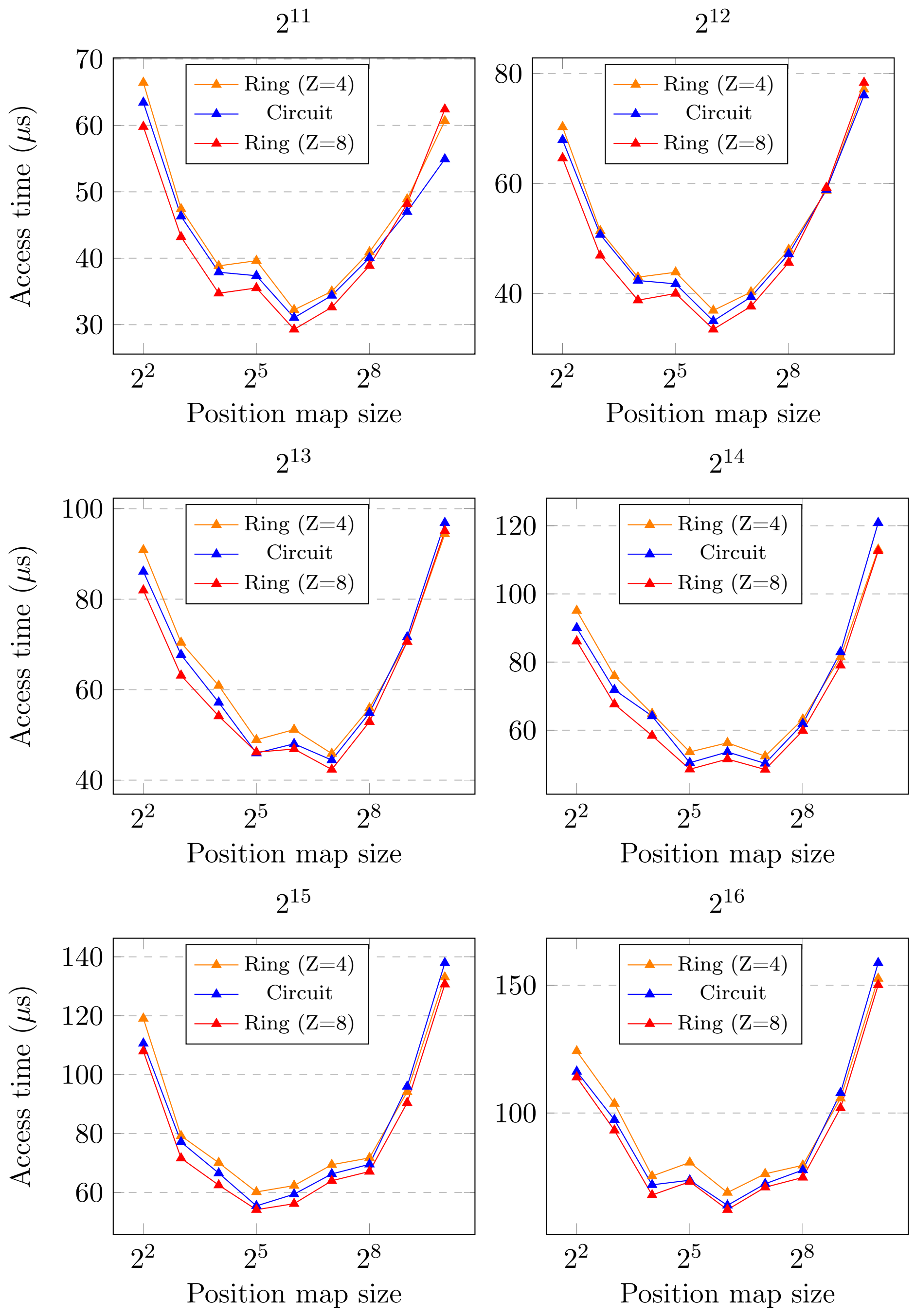

Figure A.2: Comparison of recursive position map sizes for 2^{11} to 2^{16}

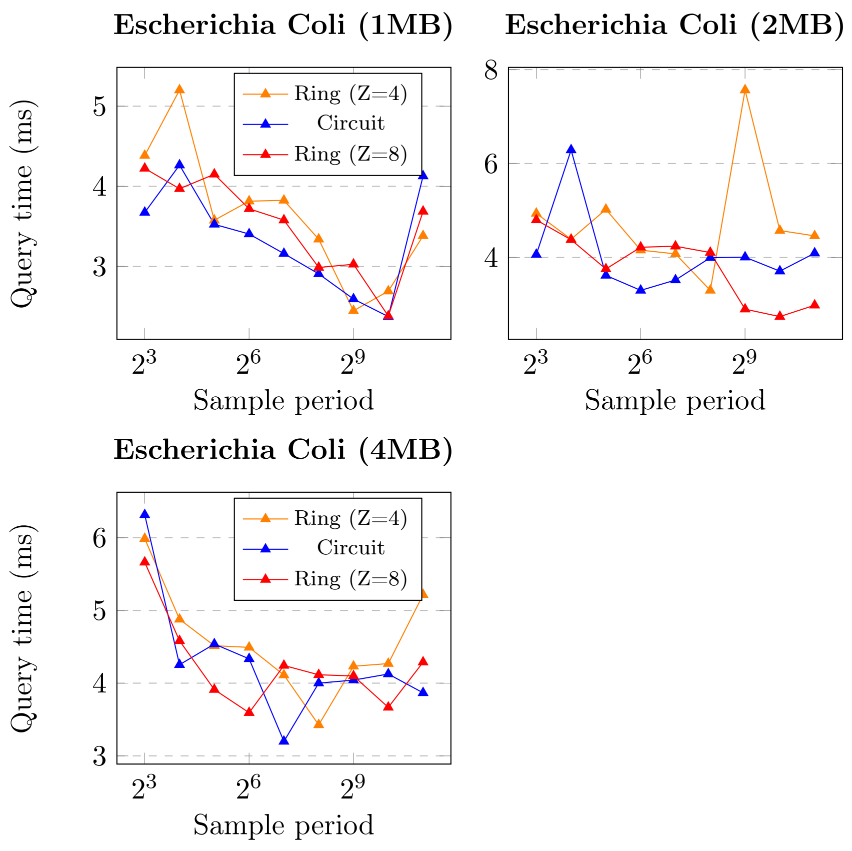

Figure A.3: Calibration of the sampling period R for `ecoli`

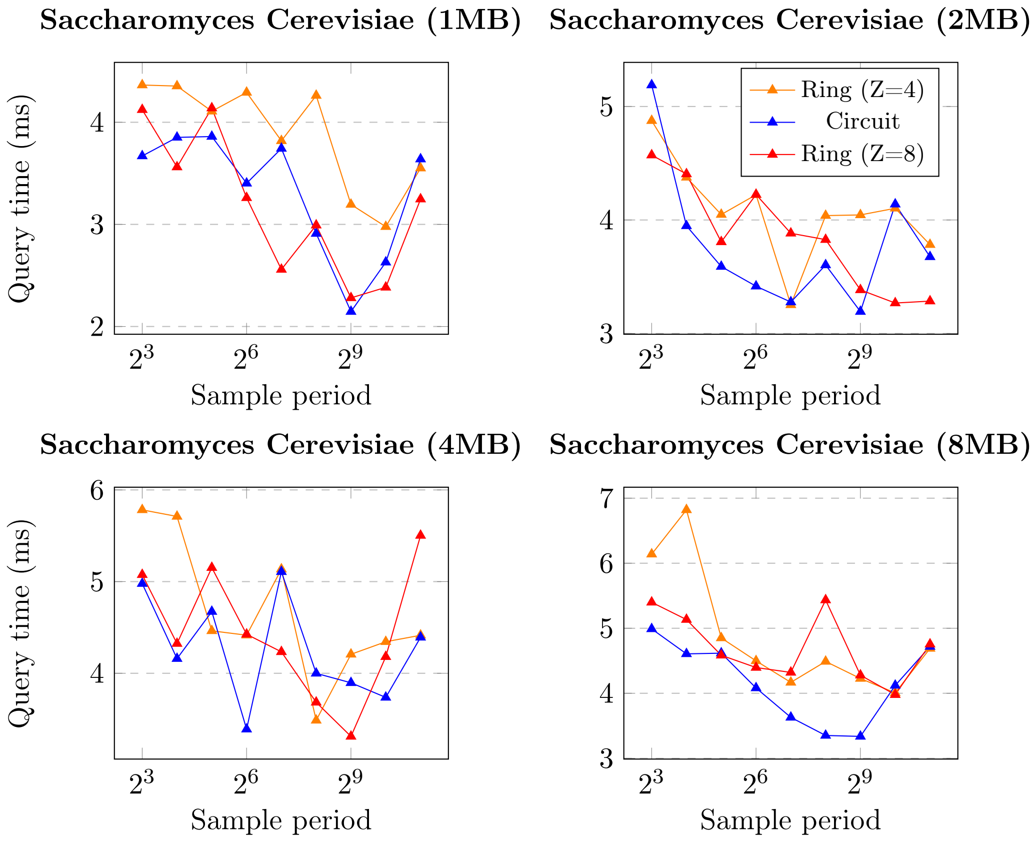

Figure A.4: Calibration of the sampling period R for `sacc`

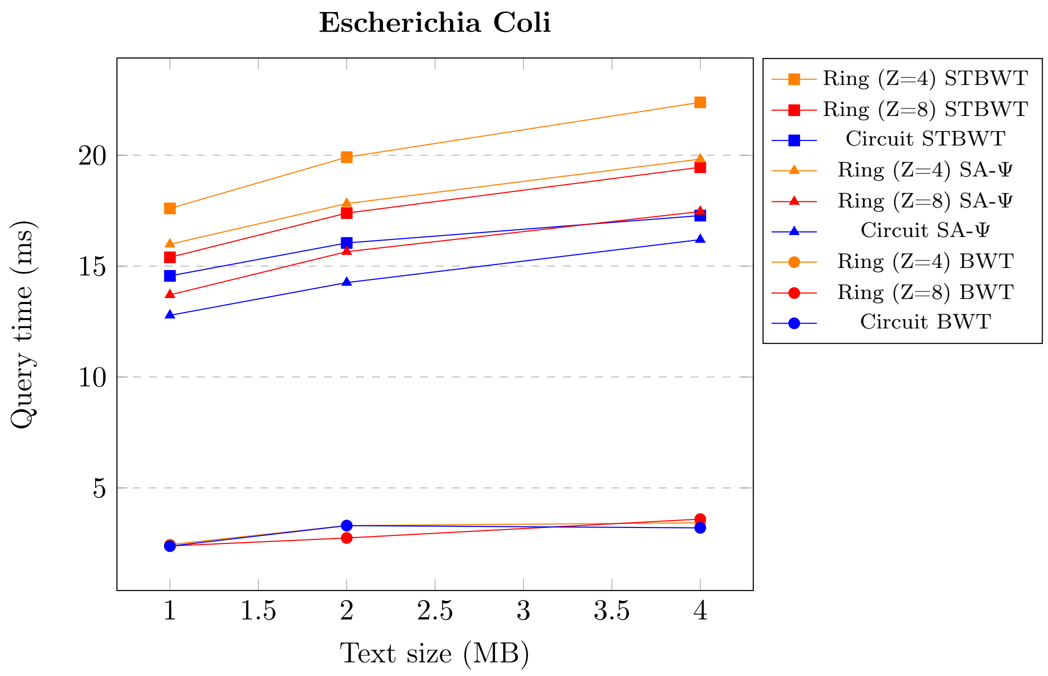

Figure A.5: Comparison of ObSQRE text indices for `ecoli`

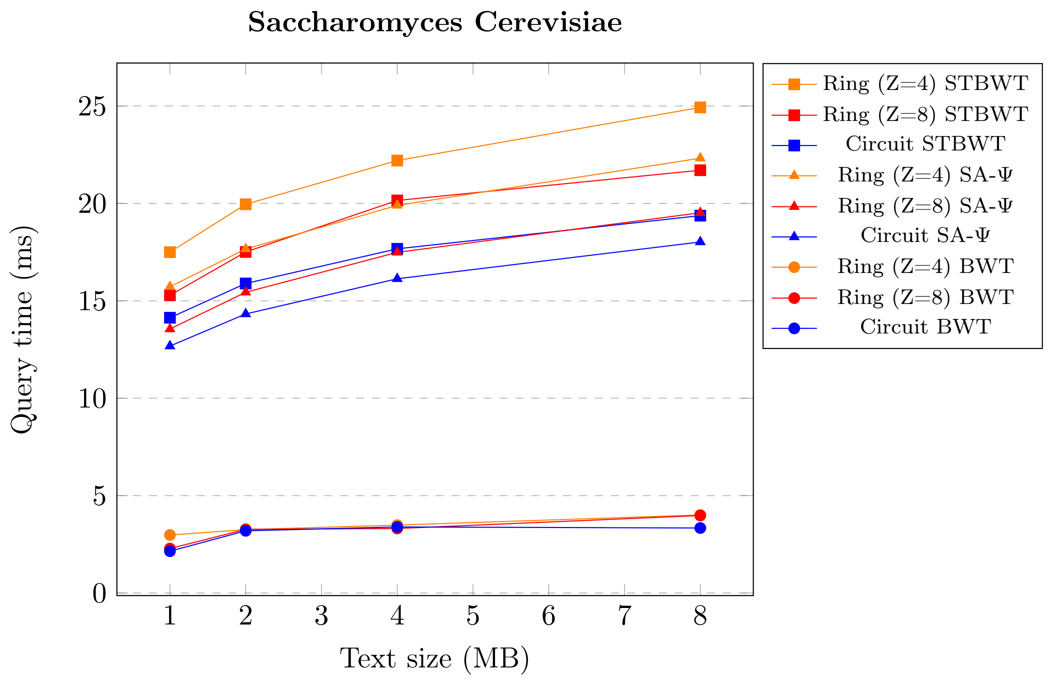

Figure A.6: Comparison of ObSQRE text indices for `sacc`

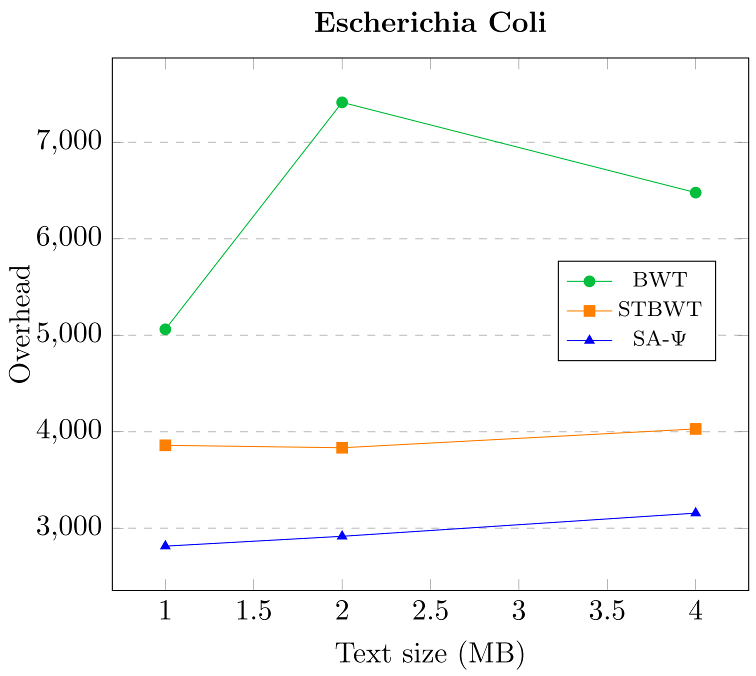

Figure A.7: Overhead of ObSQRE over baseline for `ecoli`

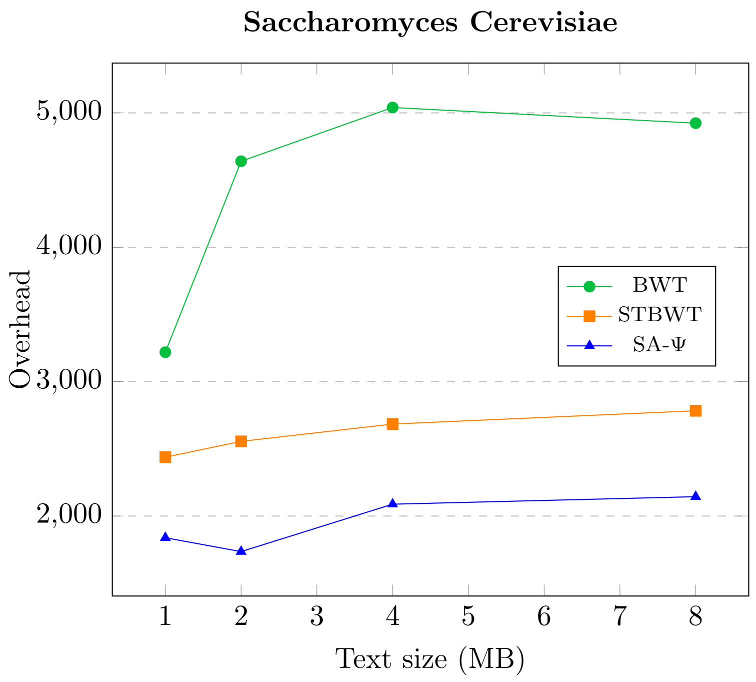

Figure A.8: Overhead of ObSQRE over baseline for `sacc`

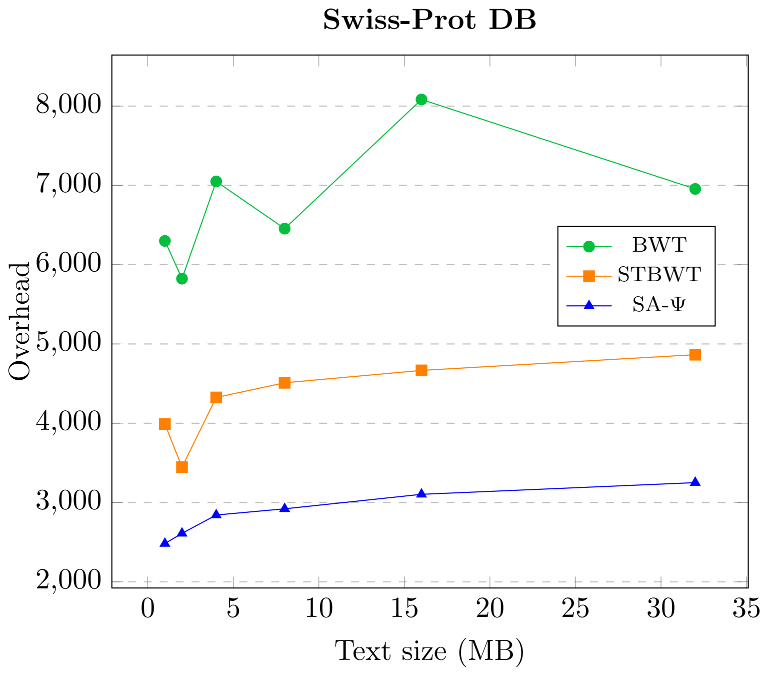

Figure A.9: Overhead of ObSQRE over baseline for `prot`

Bibliography

[1] Ghous Amjad, Seny Kamara, and Tarik Moataz. Forward and backward private searchable encryption with sgx. In *Proceedings of the 12th European Workshop on Systems Security*, EuroSec '19, pages 4:1–4:6, New York, NY, USA, 2019. ACM.

[2] 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 17)*, Vancouver, BC, 2017. USENIX Association.

[3] 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 *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, 2017. USENIX Association.

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[5] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 668–679, 2015.

[6] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client ORAM. In *26th Annual Network and Dis-*

tributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019, 2019.

[7] Melissa Chase and Emily Shen. Substring-searchable symmetric encryption. *Proceedings on Privacy Enhancing Technologies*, 2015, 06 2015.

[8] 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.

[9] Intel Corporation. Attestation service for intel software guard extensions (intel sgx): Api documentation, 2018.

[10] Intel Corporation. Code sample: Intel software guard extensions remote attestation end-to-end example, May 2018.

[11] Intel Corporation. *Intel Software Guard Extensions (Intel SGX) SDK for Linux* OS*, v2.5 edition, May 2019.

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

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

[14] Wong ED, Karra K, Hitz BC, Hong EL, and Cherry JM. The yeastgenome app: the saccharomyces genome database at your fingertips. Feb 2013. The 9th International Symposium on String Processing and Information Retrieval.

[15] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005.

[16] Christopher W. Fletcher. *Oblivious RAM: From Theory to Practice.* PhD thesis, Massachussets Institute of Technology, 2016.

[17] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. Hardidx: Practical and secure

index with sgx. In Giovanni Livraga and Sencun Zhu, editors, *Data and Applications Security and Privacy XXXI*, pages 386–408, Cham, 2017. Springer International Publishing.

[18] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. volume 7981, pages 1–18, 07 2013.

[19] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking Web Applications Built On Top of Encrypted Data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1353–1364, 2016.

[20] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, 2017. USENIX Association.

[21] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, NY, USA, 1997.

[22] B. Kaliski. Pkcs #5: Password-based cryptography specification version 2.0. RFC 2898, RFC Editor, September 2000. `http://www.rfc-editor.org/rfc/rfc2898.txt`.

[23] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Boston, third edition, 1997.

[24] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296, May 2018.

[25] Tarik Moataz and Erik-Oliver Blass. Oblivious substring search with updates. *IACR Cryptology ePrint Archive*, 2015:722, 2015.

[26] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 Data Compression Conference*, pages 193–202, March 2009.

[27] L. Ren, C. W. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious-ram. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sep. 2013.

[28] Ling Ren, Christopher 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)*, pages 415–430, Washington, D.C., 2015. USENIX Association.

[29] Cédric Van Rompay, Refik Molva, and Melek Önen. A Leakage-Abuse Attack Against Multi-User Searchable Encryption. *PoPETs*, 2017(3):168, 2017.

[30] 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, 2016.

[31] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel sgx. Cryptology ePrint Archive, Report 2017/549, 2017. `https://eprint.iacr.org/2017/549`.

[32] William Shakespeare. *The Complete Works of William Shakespeare*. Project Gutemberg, Jan 1994.

[33] Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 197–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[34] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. Internet Society, February 2017.

[35] 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*, ASIA CCS '16, pages 317–328, New York, NY, USA, 2016. ACM.

[36] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *J. ACM*, 65(4):18:1–18:26, April 2018.

[37] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, Sep 1995.

[38] The UniProt Consortium. UniProt: the universal protein knowledgebase. *Nucleic Acids Research*, 46(5):2699–2699, 02 2018.

[39] 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 *Proceedings of the 27th USENIX Security Symposium.* USENIX Association, August 2018. See also technical report Foreshadow-NG.

[40] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 850–861, New York, NY, USA, 2015. ACM.

[41] 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*, CCS '14, pages 215–226, New York, NY, USA, 2014. ACM.

[42] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015.

[43] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, 2017. USENIX Association.