



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Enforcing Security Requirements in Smart Contracts: A Decision- Making Framework

TESI DI LAUREA MAGISTRALE IN
COMPUTER ENGINEERING - INGEGNERIA INFORMATICA

Author: **Tecla Perenze**

Student ID: 971223

Advisor: Prof. Mattia Salnitri

Co-advisors: Giovanni Meroni

Academic Year: 2022-23

Abstract

Ensuring the security of smart contracts is essential for their successful execution and reliability. While blockchain technology provides a secure environment, conflicts may arise between security requirements and blockchain characteristics. This thesis tackles this challenge by determining the appropriate placement of security elements whether they should be stored on-chain or off-chain—based on their security requirements and impact on the blockchain structure. Building upon an existing algorithm, this research proposes an enhanced approach that utilizes SecBPMN2BC, a programming modeling language guiding the design of secure business processes for blockchain implementation using smart contracts. By evaluating the security requirements, the novel algorithm optimizes security elements' placement to maximize security while minimizing conflicts.

The thesis focuses on the development of two algorithms that enhance smart contract security. Both algorithms address the placement of security elements within smart contracts by using SecBPMN2BC annotations to analyze the security requirements associated with each element, considering factors such as data integrity, availability, etc. The algorithms determine whether an element should reside on-chain or in an off-chain environment.

The first is a brute-force algorithm that focuses on exhaustively evaluating the security requirements of each element within a smart contract at a global level. It systematically analyzes security annotations related to each element of the smart contract to determine whether an element should be stored on-chain or off-chain by not excluding any possible case. While this approach may require more computational resources and time, it explores all possible combinations of security considerations for each element.

The second algorithm is an optimized algorithm that aims to improve the efficiency and performance of the element placement process. By leveraging optimization strategies, the algorithm identifies the most suitable placement for elements, considering both security requirements and the impact on the blockchain structure. With this approach, the computational overhead is reduced while maintaining a high level of security.

This research combines the concepts of security annotations, SecBPMN2BC, and algorithmic analysis to provide a practical framework for the secure placement of smart contract

elements. The developed algorithms provide valuable insights and guidance for developers, ensuring that security requirements are effectively met while optimizing the performance and efficiency of smart contract execution.

Keywords: Blockchain, Smart Contract, SecBPMN2BC, Security requirements

Abstract in lingua italiana

Garantire la sicurezza degli smart contracts è fondamentale per la loro corretta esecuzione e affidabilità. Sebbene la tecnologia blockchain fornisca un ambiente sicuro, possono sorgere conflitti tra i requisiti di sicurezza e le caratteristiche della blockchain. Questa tesi affronta questa sfida determinando il posizionamento appropriato degli elementi di sicurezza, sia che debbano essere memorizzati sulla blockchain (on-chain) o al di fuori di essa (off-chain), in base ai loro requisiti di sicurezza e all'impatto sulla struttura della blockchain. Sulla base di un algoritmo esistente, questa ricerca propone un approccio migliorato che utilizza SecBPMN2BC, un linguaggio di modellazione di programmazione che guida la progettazione di processi aziendali sicuri per l'implementazione di blockchain mediante smart contracts. Valutando i requisiti di sicurezza, il nuovo algoritmo ottimizza il posizionamento degli elementi di sicurezza al fine di massimizzare la sicurezza e ridurre al minimo i conflitti.

La tesi si concentra sullo sviluppo di due algoritmi che migliorano la sicurezza degli smart contracts. Entrambi gli algoritmi affrontano il posizionamento degli elementi di sicurezza all'interno degli smart contracts utilizzando annotazioni di SecBPMN2BC per analizzare i requisiti di sicurezza associati a ciascun elemento, considerando fattori come l'integrità dei dati, la disponibilità, ecc. Gli algoritmi determinano se un elemento deve risiedere sulla blockchain o in un ambiente esterno.

Il primo è un algoritmo di forza bruta che si concentra sull'esame esaustivo dei requisiti di sicurezza di ciascun elemento all'interno di un contratto intelligente a livello globale. Analizza sistematicamente le annotazioni di sicurezza relative a ciascun elemento del contratto intelligente per determinare se un elemento deve essere memorizzato sulla blockchain o al di fuori di essa, senza escludere alcun possibile caso. Sebbene questo approccio possa richiedere più risorse computazionali e tempo, esplora tutte le possibili combinazioni di considerazioni di sicurezza per ciascun elemento.

Il secondo algoritmo è un algoritmo ottimizzato che mira a migliorare l'efficienza e le prestazioni del processo di posizionamento degli elementi. Sfruttando strategie di ottimizzazione, l'algoritmo individua il posizionamento più adatto per gli elementi, considerando

sia i requisiti di sicurezza che l'impatto sulla struttura della blockchain. Con questo approccio, si riduce l'onere computazionale pur mantenendo un alto livello di sicurezza.

Questa ricerca combina i concetti di annotazioni di sicurezza, SecBPMN2BC e analisi algoritmica per fornire un quadro pratico per il posizionamento sicuro degli elementi degli smart contracts. Gli algoritmi sviluppati forniscono preziose informazioni e orientamenti per gli sviluppatori, garantendo che i requisiti di sicurezza siano soddisfatti in modo efficace, ott

imizzando al contempo le prestazioni e l'efficienza dell'esecuzione degli smart contracts.

Parole chiave: Blockchain, Smart Contract, SecBPMN2BC, Security requirements

Contents

Abstract	i
Abstract in lingua italiana	iii
Contents	v
1 Introduction	1
2 State of Art	5
2.1 Optimizing Smart Contract Code for Storage Decisions	5
2.2 Enhancing Smart Contract Security	8
3 Baseline	11
3.1 Securing Smart Contracts with SecBPMN2BC	11
3.2 On-chain and Off-chain Elements in Securing Business Processes	12
3.3 SecBPMN2BC	12
3.3.1 Security Requirements	13
3.3.2 Privity	14
3.3.3 Enforceability	15
3.4 SecBPMN2BC	15
3.4.1 Type of Blockchain	16
3.4.2 Rules for Property Combination	17
3.5 Running Example	18
4 Algorithmic Framework	21
4.1 Global Enforcement	22
4.1.1 "Possible" enforcement level	23
4.2 Common initial part of the proposed algorithms	25
4.2.1 Structure of the algorithm	31

5	Bruteforce strategy	33
5.1	Propagate up	33
5.1.1	Pseudocode Propagate-up Bruteforce	39
5.2	Propagate down	45
5.2.1	Pseudocode PropagateDown Bruteforce	50
6	Optimized strategy	53
6.1	Propagate up	53
6.1.1	Pseudocode Propagate-up Optimized	57
6.2	Propagate Down	61
6.2.1	Pseudocode PropagateDown Optimized	64
7	Validation	67
7.1	Experimental Setup	67
7.2	Evaluated Properties	68
7.3	Results Analysis	69
7.4	Discussion of Findings	75
8	Conclusions and future developments	77
	Bibliography	79
	List of Figures	83
	List of Tables	85
	Acknowledgements	87

1 | Introduction

Smart contracts are blockchain-based algorithms that are executed when certain criteria are met, which has the advantage of eliminating the need for intermediaries and third parties in transaction processes, reducing costs. Indeed, in order to define how a process should be structured they provide a set of rules and instructions, enabling the enforcement of agreements between parties, such as asset ownership transfer, service delivery, or fee payment[34]. Once deployed, smart contracts are transparent and immutable and it makes their results tamper-proof and unchangeable.

The automated nature of smart contracts offers a range of advantages over traditional contracts, including improved efficiency and cost-effectiveness [3]. These features provide transparency and immutability that traditional contracts cannot match, making them increasingly popular in various industries. In the supply chain management domain, for instance, smart contracts can be used to automate product traceability and authenticity verification from the factory to the end consumer [7][37].

However, smart contracts must adhere to certain security requirements to be effective [30] These programs are based on algorithms and can pose risks if incorrectly designed and implemented. Therefore, security is extremely important as security vulnerabilities may cause significant financial and legal consequences for the parties involved. It has been reported that a high percentage of informatic attacks in recent years were focused on smart contracts, resulting in a loss of cryptocurrency. In the world of decentralized finance (DeFi), there have been notable incidents that highlight the risks and vulnerabilities associated with smart contracts. The bZx hack in 2020, the DAO attack in 2016, Uniswap exploits, and the Harvest Finance attack all serve as reminders of the importance of security measures and diligent code auditing. These incidents resulted in significant financial losses, emphasizing the need for ongoing advancements in DeFi security practices to safeguard user funds and enhance the overall resilience of the ecosystem [25]

To mitigate these risks, Blockchain technology provides a secure and transparent environment for the execution of smart contracts. The decentralized and immutable nature of the blockchain ensures that once a smart contract is deployed, it becomes part of a

tamper-proof and transparent ledger that is maintained by a network of nodes [17]. The transaction is verified by each node, which also updates the blockchain to reflect the modifications made by the smart contract. As the network must agree on any modifications to the contract, it is extremely difficult to manipulate or commit fraud with it.

However, although blockchain technology offers a high level of security for smart contracts, it may not always be able to fully satisfy all security requirements. In truth, there may be inconsistencies between some security criteria and aspects of blockchain technology. Therefore, to ensure the utmost security of smart contracts, it is imperative to determine which data elements should be stored on-chain or off-chain based on their security requirements and the potential impact on the blockchain structure[11]

While critical data elements should be stored on-chain for maximum transparency and security, it is crucial to consider the potential conflicts with the blockchain structure and the increased risk of security breaches that may arise from storing certain data on-chain. For example, conflicts may arise between confidentiality and availability if a system is designed to be highly secure but not easily accessible to authorized users. Similar issues might develop between authenticity and integrity if the technology supports anonymous transactions. Thus, in order to spot and resolve conflicts and make sure that the blockchain system is safe and effective at fulfilling the security requirements of smart contracts, it is essential to carefully evaluate the security requirements of each data element and its potential impact on the blockchain structure. [31]

To this purpose, the primary goal of this thesis is to identify which elements of a given process can be stored on-chain, rather than off-chain. The goal of the study is to assess the process' security needs and identify the best components that can be stored utilizing blockchain technology. It is presented a novel approach to improve an already existing algorithm [13] that utilizes SecBPMN2BC, a technique that guides process modelers and security specialists in designing secure business processes that are suitable for implementation on a blockchain using smart contracts.

The thesis begins with an introduction that surveys the existing literature on smart contract optimization, providing a solid foundation for security fields. It introduces other algorithms and frameworks that share the same objective but employ distinct approaches, offering a broader perspective on diverse strategies. The baseline section presents the original algorithm and SecBPMN2BC's basic structure, while the running example description helps provide a clearer vision of the algorithms' flow. The following chapters detail two algorithms: the brute-force version and the optimized version. Both chapters have a similar structure, beginning with functionalities and a running example.

The pseudocodes are detailed, and a validation chapter is introduced to highlight the differences between the two versions and their benefits. Test cases are meticulously built to test both versions and ensure they achieve the expected results. The thesis concludes with a conclusion and future development paragraph.

2 | State of Art

The proliferation of blockchain technology and smart contracts has increased the demand for secure data processing and storage. One of the critical decisions developers face is determining whether certain contract components should be stored on-chain or off-chain. This decision is critical because storing data on-chain can be expensive and slow down the blockchain's performance while storing data off-chain can increase the risk of tampering. Therefore, developers must strike a balance between security and scalability. Ensuring the security of smart contracts is of utmost importance to prevent attacks such as reentrancy and overflow attacks, which can result in significant financial losses [6].

2.1. Optimizing Smart Contract Code for Storage Decisions

In the realm of smart settlement development, making sure the security of code is of paramount significance. Optimizing clever agreement codes for storage selections performs a crucial position in enhancing the security posture of decentralized packages. A modern focus of research inside the discipline of smart contracts is indeed the optimization in their code to determine whether or not sure components have to be saved on-chain or off-chain. Several techniques have been proposed to deal with this difficulty, which include machine learning, dynamic evaluation, and static evaluation, as discussed by Deeple et al. [36]. Dynamic analysis includes tracking smart contracts during use to discover ability security vulnerabilities and make greater knowledgeable selections about information garage location. In assessment, static evaluation includes reading clever agreement code prior to deployment to discover capacity safety weaknesses [30] and make optimal storage decisions. Machine learning-based techniques, on the other hand, involve training models beyond smart contracts transactions too expecting the ideal storage vicinity for brand-new transactions. By using these tactics, researchers aim to broaden secure and stable smart contracts that could perform on both on-chain and stale-chain storage systems.

In their survey paper, "*Enhancing Smart-Contract Security through Machine Learning: A*

Survey of Approaches and Techniques," Jiang et al.[22] explore numerous system getting to know-based totally methods to decorate the security of smart contracts. The paper gives a complete evaluation of present research on the usage of gadget mastering to hit upon vulnerabilities and mitigate dangers associated with smart contracts. The paper gives a whole assessment of existing research on using system mastering expertise to discover vulnerabilities and mitigate dangers related to smart contracts. While device getting to know can help optimize the selection-making technique for on-chain and off-chain storage, it isn't a fail-secure solution. Indeed, they can be liable to biases and require fantastic amounts of statistics to educate efficaciously. Additionally, those methods may not be capable of discovering all ability safety vulnerabilities, and the models themselves may be vulnerable to assaults. As such, gadget-studying-based answers need to be used together with exclusive measures to ensure the safety and integrity of smart contracts.

The field of optimizing smart contract security and storage decisions is a complicated and hastily evolving area of research. As the call for blockchain technology maintains to grow, the want for steady processing and storage becomes more and more vital. One sizeable factor that researchers and developers should bear in mind is interoperability among distinct blockchain networks. However, at the same time as cross-chain solutions provide promising opportunities for seamless exchange and information transfer between chains, they'll not be sufficient to cope with all of the demanding situations and considerations associated with smart contract protection and storage choices.

These researches are not only extended to machine learning but technique as Cross-chain interoperability are taken into consideration. When talking about this field, in literature numerous research paper talks about the integration of smart contract on the blockchain, concerning [15] Cross-chain solutions focus on enabling interoperability by facilitating the transfer of assets and data across different blockchain networks. These solutions include techniques such as atomic swaps, bridge protocols, and interoperability frameworks. They aim to create a unified ecosystem that allows smart contracts to interact with multiple chains, expanding the functionalities and possibilities of decentralized applications. While cross-chain solutions contribute to enhancing the overall efficiency and connectivity of blockchain networks, they may fall short in addressing specific security and storage concerns. It is important to recognize that cross-chain communication introduces additional complexities and potential vulnerabilities. The security of smart contracts becomes more intricate when they interact with multiple chains, as each chain may have its own unique characteristics, consensus mechanisms, and security protocols. As the demand for blockchain technology continues to grow, the need for secure and efficient data processing and storage becomes increasingly crucial. One significant aspect that researchers

and developers must consider is interoperability between different blockchain networks. However, while cross-chain solutions offer promising opportunities for seamless communication and data transfer between chains, they may not be sufficient to address all the challenges and considerations related to smart contract security and storage decisions.

To cope with the specific approach of figuring out whether an element of a smart contract must be placed on-chain or off-chain, numerous initiatives and initiatives had been actively running on developing answers in this domain.

One notable example is Polkadot, a multi-chain platform designed to enable cross-chain communication and interoperability. Polkadot allows different blockchain networks to connect and share information through a unified protocol, enabling seamless interaction between chains[35]. This platform facilitates the decision of whether to put elements on the chain or off-chain through its parachain infrastructure. Parachains are specialized chains connected to the Polkadot Relay Chain, which acts as the main network for coordination and consensus[26]. These Parachains can have their own rules, governance mechanisms, and functionalities, however, are nevertheless interoperable with different Parachains and the Polkadot community as a whole. Consequently, it is possible to create and deploy smart contracts on specific Parachains through this platform, this gives developers the flexibility to determine which elements of the smart contract should be executed on the chain or off-chain, based on various considerations. Developers can leave elements on-chain to achieve a high level of transparency, immutability, or decentralized execution, or execute them off-chain to improve performance and scalability. What is important to note is that Off-chain workers can execute functions outside the blockchain network, but are still connected to the parachain and can interact with on-chain data. Thus it is crucially important to remember that developers should not have the entire ability to decide whether to deploy components on-chain or off-chain, rather, it should be based on an analysis of the smart contract's security needs. This evaluation manner makes it positive that the proper steps are taken to shield the agreement's integrity and confidentiality and to make sure they're steady with the general protection desires. Hence, as opposed to simply counting on developer judgment, the selection-making technique has to be based on an intensive grasp of the safety wishes.

This evaluation manner makes it positive that the proper steps are taken to shield the agreement's integrity and confidentiality and to make sure they're steady with the general protection desires. Hence, as opposed to simply counting on developer judgment, the selection-making technique has to be based on an intensive grasp of the safety wishes.

2.2. Enhancing Smart Contract Security

Existing tools and frameworks, such as Remix [19], Solidity [20], and Truffle [21], are available to optimize smart contract code for decision-making about on-chain and off-chain storage. However, these technologies may not always provide optimal solutions in dynamic situations such as changing network congestion and petrol prices.

Further research is needed to improve the decision-making process for on-chain and off-chain storage of smart contract data, with additional security requirements and optimization strategies being explored. Some algorithms have been developed to determine the security of the smart contract implemented on the blockchain; however, the advantages and disadvantages of these approaches must be carefully considered. Nevertheless, the contributions of these developments have been instrumental in advancing the field of smart contract security and storage optimization.

The *Gaspar* [8] is a framework for trying out the safety of smart contracts. It gives a complete trying-out method that covers a huge range of security vulnerabilities, automates testing, is open-supply, customizable, and calls for giant computing assets. Nevertheless, this method may not be fail-secure in identifying all ability safety vulnerabilities in smart contracts, as it's far complicated and requires a technical understanding to apply successfully, it's far resource-extensive and is depending on Solidity, that is the programming language utilized by the Ethereum blockchain. Therefore, it can now not appropriate for testing smart contracts written in different languages or for different blockchain systems.

Securify [33] Is a valuable safety tool that could considerably enhance the security of smart contracts. The tool is designed to tackle the venture of figuring out ability security vulnerabilities in smart contracts thru complete security evaluation through the use of static and dynamic analysis techniques. By conducting an intensive analysis of the contract's code, Securefy can detect any safety flaws and generate certain reports on the issues located, supplying recommendations on the way to cope with them. In addition, Securefy gives non-stop tracking to discover any suspicious interest and provides signals for ability protection issues, permitting developers to take on-the-spot action. The use of Securefy can help enhance the security of smart contracts, minimizing the danger of hacks, exploits, and insects. However, it's miles important to observe that the use of Securefy can also come with a cost that can consist of economic prices, together with licensing costs or subscription fees for accessing the service, which can be an obstacle for some developers[1]. Furthermore, the tool's compatibility with certain smart contract platforms or languages may be limited, which can result in false positives or reduced effectiveness for certain developers. Despite these potential limitations, Securefy remains

a valuable tool for enhancing the security of smart contracts and mitigating the risk of security breaches.

Formal verification instead is a rigorous approach for verifying the correctness of smart contracts by using mathematically proving properties approximately their behavior. It entails the usage of formal common sense and automated gear to analyze the contract's code and confirm houses along with the absence of vulnerabilities and adherence to favored specifications. Formal verification techniques were proposed as a manner to beautify the safety of smart contracts and make informed selections approximately on-chain and off-chain storage.

Research studies, such as the work by Bhargavan et al. [5], Have explored the software of formal verification strategies for smart contracts. This research demonstrates the capacity of formal strategies to come across vulnerabilities and make certain the correctness of contract execution. However, certain limitations prevent formal verification from fully achieving the scope of ensuring safety solely via on-chain or off-chain storage. First, formal verification strategies often require huge expertise and assets to use efficaciously. The procedure entails modeling the agreement, specifying houses, and utilizing formal verification gear, which may be complicated and time-ingesting with the result of restricting the accessibility of formal verification to developers with specialized expertise or hindering its adoption in practical development eventualities. Second, in preference to reading the full contract code, formal verification procedures are frequently higher applicable for studying smaller, essential factors of the settlement. Owing to the complexity of smart contracts and the scalability constraints of formal verification, doing comprehensive analysis on huge and complex contracts can be impractical. As a result, while formal verification might provide extensive insights into the safety of positive additives, it may no longer provide a whole solution for selecting in which all settlement parts need to be stored[2]

The paper "*Hawk: The blockchain model of cryptography and privacy-preserving smart contracts*" by Kosba et al.[12] introduces the Hawk framework, which focuses on ensuring privacy in smart contracts through the use of cryptographic techniques. The framework incorporates various cryptographic protocols such as zero-knowledge proofs, secure multiparty computation, and encryption to address the crucial issue of privacy in smart contracts. The Hawk framework hides clever settlement inputs and outputs, permitting verification of correctness and integrity at the same time as preserving strict confidentiality of sensitive records. The integration of cryptographic protocols enhances the safety and protection of sensitive statistics in smart contracts. However, it's far more important to be aware that the paper acknowledges the potential overall performance and computational trade-offs that can get up from using privacy-retaining techniques. Although the

paper provides a basis for evaluating the level of privacy safeguards supplied by means of the Hawk framework, it does not discuss in detail the particular issues and challenges related to in-chain and out-of-chain storage selections. It additionally lacks a comparative evaluation of alternative privacy procedures or techniques. In addition, the paper no longer provides empirical reviews or case research to validate the practicality and effectiveness of the Hawk framework in real-global eventualities. While the Hawk framework helps deal with privacy issues in smart contracts, its barriers in terms of the scope of storage decisions and lack of comprehensive evaluation are motives for similar research.

Additional studies should encompass a broader range of factors, including scalability [2], cost [1], performance, and potential risks associated with data tampering or breaches [9], to provide a more comprehensive understanding of security considerations and informed storage decisions in smart contracts.

The work "*Scalable and Privacy-preserving Design of On/Off-chain Smart Contracts*" by Li et al. [14], which tackles the combined difficulties of scalability and privacy in the design of on/off-chain smart contracts, highlights another intriguing strategy. It promises to offer technologies that let smart contracts manage big transactions while safeguarding the privacy of private information. Given the rising demand for blockchain-based applications and the requirement to support a large number of transactions, the article explores the significance of scalability in smart contracts. It investigates methods to increase the scalability of smart contracts, enabling more effective execution and lower computing cost, such as off-chain processing and sharding. The research highlights the need for privacy protection in smart contracts in addition to scalability. In order to safeguard sensitive information during contract execution, it examines privacy-enhancing strategies like as zero-knowledge proofs, secure multi-party computing, and selective disclosure. These methods make sure that only pertinent information is shared while protecting the privacy of other contract-related data. It has to be emphasized that the research no longer passes into excellent intensity on how the advised scalable and privacy-retaining approach is sincerely applied. Instead of offering in-depth technical solutions or empirical judgments, it could supply an excessive degree evaluate and conceptual framework. While the paintings advance information on the issues and potential fixes for scalable and privateness-preserving smart contracts, extra investigation and actual-international programs are required to check and enhance the suggested structure.

3 | Baseline

The chapter provides a top-level view of the modern-day today's approach hired to deciding the suitable execution location (on-chain or off-chain) for specific additives inside a smart contract. This decision is based on the corresponding safety necessities related to each factor. Additionally, there is an introduction to the SecBPMN2BC modeling language, which serves as a graphical representation of smart contracts and permits the specification of their safety requirements.

This chapter is based on an algorithm proposed in the existing literature [18]. This algorithm aims to identify local optimal solutions for individual elements of a smart contract, taking into consideration factors such as their security requirements, privity, and enforceability. However, it is important to note that this algorithm has a limitation, as it does not guarantee a globally optimal solution for the entire smart contract. Thus, a novel optimization approach to address this limitation. This approach aims to achieve a globally optimal solution that encompasses the entire smart contract.

3.1. Securing Smart Contracts with SecBPMN2BC

SecBPMN2BC is an extension of the widely used Business Process Model and Notation (BPMN) [32] architecture designed to address security concerns in modeling and implementing blockchain technology business processes[29]. The primary objective of SecBPMN2BC is to ensure that smart contracts are secure and tamper-proof by defining security requirements at the process level, including access control, data confidentiality, and integrity. Furthermore, SecBPMN2BC also considers potential conflicts between data minimization and security requirements in business process models, providing a way to detect and resolve such conflicts [28]. Organizations may utilize SecBPMN2BC to create more dependable and secure smart contracts, which are essential for maximizing the promise of blockchain technology.

3.2. On-chain and Off-chain Elements in Securing Business Processes

On-chain elements of a smart contract generally encompass the contract code, contract state, and transaction statistics. These on-chain actors are handled via the nodes in the blockchain community and are seen by all individuals in the community [23] [27].

Off-chain elements of a smart contract can consist of any facts or computation that is not saved on the blockchain. In particular, off-chain transactions are focused on exchanges that take place outside of the Blockchain and can be finished using a variety of methods. What is crucial is that all the parties must consent to the transfer, and a third party must then certify the transaction. Off-chain transactions are quick and instantaneous and don't include the extra fees that on-chain transactions do [10].

The choice of which element of the smart contract has to be stored on-chain or off-chain depends on a variety of factors, along with the complexity of the computation, the quantity of information that needs to be saved or accessed, and the desired stage of transparency and safety. In general, the parts of a smart contract that involve sensitive data or require a high level of security should be kept on-chain, while other parts of the contract that are less critical can be placed off-chain. For example, sensitive information as it can be private keys, personal data or critical business logic that determines the behavior of the smart contract, such as payment or asset transfer functions that require secure processing must be secured on the blockchain. Instead, any kind of data that does not involve sensitive information i.e. Non-critical business logic or calculations that do not affect the outcome of the contract can be stored off-chain [24].

3.3. SecBPMN2BC

SecBPMN2BC (Secure Business Process Modeling Notation 2 Blockchain) focuses on secure business process modeling and design. To determine which parts of the smart contract can go on-chain or off-chain,[31] it is important to:

- prioritize the essential security requirements
- examining the security concerns involved
- select components that can off-chain without threatening security
- identify non-critical operations or data that don't need blockchain security

- choose which components must be executed on the blockchain
- install suitable security measures for both on-chain and off-chain components.

An improved version of the modeling language SecBPMN2BC addresses this concern by incorporating security measures. The following sections will provide a more detailed discussion of this topic.

3.3.1. Security Requirements

As mentioned above, SecBPMN2 is an extension of BPMN 2.0 that includes a comprehensive set of security requirements. It was chosen as the baseline for SecBPMN2BC because of its widespread use and rich security features. It is constructed with security requirements that are derived from the BPMN elements they are associated with[18]. Annotations are suffixed with specific codes (such as "Act" for activity, "DO" for data object, and "MF" for message flow) to distinguish their meaning based on the linked BPMN element. All security annotations are shown in Figure 3.1.










 Auditability	 Separation of duties
 Authenticity	 Bind of duties
 Availability	 Non Delegation
 Integrity	 Privacy
 Non Repudiation	

Figure 3.1: Graphical annotations for security annotations of SecBPMN2BC [18]

- **Authenticity.** This security requirement refers to the assurance that the identity of a user or system is valid and verified. Authenticity is established through a process of authentication that verifies the identity of the user or system.
- **Availability.** Is a security requirement that ensures that information and systems are accessible and usable by authorized users when they need it. This includes measures to prevent and mitigate unauthorized access, system failures, or other disruptions.
- **Integrity.** Refers to the assurance that information and systems have not been tampered with, altered, or modified in an unauthorized way. This includes measures

to protect against unauthorized access or modification, as well as to ensure the accuracy and completeness of the information.

- **Non-repudiation.** Is a security requirement that ensures that a user cannot deny having performed a specific action, such as sending a message or making a transaction. This is typically achieved through the use of digital signatures or other cryptographic techniques.
- **Separation of Duties.** Is a security requirement that ensures that no single user or system has complete control over a critical process or system. This is achieved by dividing responsibilities among multiple users or systems to prevent conflicts of interest or potential misuse.
- **Binding of Duties.** Is a security requirement that ensures that users are only authorized to perform specific tasks that are relevant to their roles and responsibilities. This helps to prevent unauthorized access and misuse of information and systems.
- **Non-delegation.** Is a security requirement that ensures that users are not authorized to delegate their access rights or responsibilities to other users. This helps to prevent unauthorized access and misuse of information and systems.
- **Privacy.** Is a security requirement that ensures that personal information is protected and used appropriately. This includes measures to prevent unauthorized access, use, or disclosure of personal information, as well as to ensure that personal information is collected and used in compliance with relevant laws and regulations.

3.3.2. Privity

Privity spheres have the ability to statically or dynamically limit read data access. Figure 3.2 presents an overview of the limitations brought about by the privity requirements on connected data items. Only the public privity sphere is natively supported on public blockchains using unencrypted on-chain data, and all other privity levels are violated. Private blockchains are capable of supporting both public and private privity-spheres out of the box, but they need extra tools like channels to do so. Due to the possibility that the same data values may need to be written to several separate channels, the strong dynamic sphere is particularly important for private blockchains with channels.






	Privity - public
	Privity - static
	Privity - private
	Privity - strong dynamic
	Privity - weak dynamic

Figure 3.2: Graphical annotations for Privity of SecBPMN2BC-ML [18]

3.3.3. Enforceability

Enforceability refers to the ability of process models to specify requirements that impose restrictions on data objects accessible by specific activities. By utilizing the Enforceability annotation, shown in Figure 3.3, in conjunction with appropriate gateways, process models can define these requirements. Public blockchains, which store input data transparently on-chain or in a digest form, naturally support public enforceability. Private blockchains with a significant number of nodes not controlled by process participants can also enable public enforceability. Emulating cross-channel transactions on the main chain can be achieved through techniques like voting protocols and zero-knowledge proofs. In certain cases, the storage of data objects on different channels may be necessary to fulfill privity requirements, and the strong dynamic sphere can involve multiple instances of the same data item stored on several channels.





	Enforceability of the control-flow
	Enforceability of decisions - public
	Enforceability of decisions - private
	Enforceability of decisions - user defined

Figure 3.3: Graphical annotations for Enforceability of SecBPMN2BC-ML [18]

3.4. SecBPMN2BC

The decision of which processes should be executed on-chain and which should rely on off-chain tools and approaches should be carefully considered when creating a blockchain-

based application. The following process components impact this choice [18]:

- Sequence of activities, pools and lanes, and message flows define the process structure. If this structure is defined on the blockchain, it can be encoded with a smart contract that monitors the process execution by emitting events when activities should be executed and receiving notifications when they start or finish. However, if the structure is defined off-chain, a traditional Business Process Management System (BPMS) manages the process execution.
- Activities refer to a single unit of work that needs to be performed and can be classified into fully automated, semi-automated, and manual activities. When fully automated activities are executed on the blockchain, the smart contract can include instructions for their execution, ensuring that the activity is performed exactly as expected. However, if fully automated activities are executed off-chain, an external software, such as a web service, executes them. Semi-automated and manual activities require real-world resources, such as machines or humans, and cannot be executed on-chain.
- Data objects or messages represent the data being manipulated by the process. If stored on the blockchain, smart contracts govern read and write operations, as well as the validation mechanisms to ensure data correctness. However, if data is stored off-chain, external applications, such as a Database Management System (DBMS), must manage both storage and validation. It's important to note that data objects may have different security properties depending on their state.

3.4.1. Type of Blockchain

When designing a blockchain-based solution, it is important to consider the type of blockchain used. Public blockchains are decentralized, allowing anyone to access information, while private blockchains are governed by a central authority or a select group of members. To specify these decisions in the SecBPMN2BC specifications, the following properties have been introduced:

- `OnChainModel`. This property specifies whether the execution logic will be handled on-chain via smart contracts (if set to true) or off-chain (if set to false).
- `OnChainExecution`. This property can be set to true or false for activities. If set to true, the activity will be executed on-chain via a smart contract, and if set to false, it will be executed off-chain. However, it is important to note that those activities that require human decision-making or any actions that cannot

be automated. are typically not suitable for on-chain execution. Therefore, it is more common to set the property to false for such activities to indicate off-chain execution.

- **OnChainData.** This property specifies how data associated with a message or data object state will be stored and validated. It can be set to unencrypted, encrypted, hash (which stores only the digest of the data on-chain), or none (which stores the data entirely off-chain).
- **BlockchainType.** This property specifies whether the on-chain portion of the process will be executed on a public or private blockchain.

3.4.2. Rules for Property Combination

According to the security annotations, previously discussed in Section 3.3.1, the process component type, and the kind of blockchain being utilized, it is possible to derive rules for obtaining the best combination of security requirements. Specifically, when analyzing each rule, the associated process elements are determined. For each element that meets the criteria of the rule, various sets of property values are identified. These sets encompass both the properties of the process element itself, as well as those of its parent or related elements. Eventually, each set of properties is given a label indicating the degree to which the blockchain's security annotation enforces the rule. The labels are "native" when the blockchain enforces the property natively, "possible" when it enforces it partially, and "no enforcement" when it provides no help in enforcing the property. In Figure 3.4, a comprehensive list of all blockchain enforcement rules for SecBPMN2 security annotations is shown.

	OnChain Model	OnChain Execution	Output Label		OnChain Model	OnChain Data	Output Label		OnChain Model	OnChain Data	Output Label
AuthenticityAct	any any	true false	native no enf.	AuthenticityDO	any any any any	unencrypted encrypted digest none	native native no enf. no enf.				
AuditabilityAct	any any	true false	native possible	AuditabilityDO	any any any any	unencrypted encrypted digest none	native native no enf. no enf.	AuditabilityMF	any any any any	unencrypted encrypted digest none	native possible no enf. no enf.
AvailabilityAct	any any	true false	native no enf.	AvailabilityDO	any any any any	unencrypted encrypted digest none	native native no enf. no enf.	AvailabilityMF	any any any any	unencrypted encrypted digest none	native possible possible possible
IntegrityAct	any any	true false	native possible	IntegrityDO	any any any any	unencrypted encrypted digest none	native native native no enf.	IntegrityMF	any any any any	unencrypted encrypted digest none	native native native no enf.
NonRepAct	any any	true false	native possible					NonRepMF	any any any any	unencrypted encrypted digest none	native native native possible
NonDelAct	any any	true false	native possible								
BoDPool (Act)	true true false	true false any	native possible no enf.	BoDPool (DO)	true true true false	unencrypted encrypted digest none any	native native possible no enf.				
SoDPool (Act)	true true false	true false any	native possible no enf.	SoDPool (DO)	true true true false	unencrypted encrypted digest none any	native native possible no enf.				

Figure 3.4: Blockchain enforcement rules for SecBPMN2BC security annotations [18]

3.5. Running Example

Throughout the chapter, a running example is utilized to illustrate the concepts discussed. The example involves a smart contract process that incorporates a Ride-sharing process. By exploring this scenario, readers gain practical insights into the challenges and considerations involved in determining the suitable execution location (on-chain or off-chain) for specific smart contract components, based on their corresponding security requirements.

This SecBPMN2BC smart contract is tailored specifically for a ride-sharing platform. It pursues to streamline the approaches of experience request control, driving force matching, payment authorization, and remarks series. By employing SECBPMN2BC, a comprehensive and established technique is taken to model the various pools, duties, records gadgets, and stop activities worried inside the ride-sharing method.

The ride-sharing smart contract includes some operations that assists with the coordination and execution of journey requests, driver assignment, payment authorization, and

feedback collection. Simultaneously, the driver, belonging to the "Driver" pool, receives the ride request and confirms its availability by accepting the assigned ride through the "Confirm Ride" task. This ensures that the driver is actively involved in the ride undertaking process.

Upon acceptance, the user and driver continue with the ride, with the driver updating the ride completion status and their availability for future ride requests using the "Complete Ride and Update Driver Status" task in the "Driver" pool. This enables better control of driver resources and availability. The payment procedure is commenced simultaneously within the "Payment" pool. This job validates the ride's final touch and the accuracy of payment records, ensuring a safe and dependable transaction. The "Payment Data" object contains the relevant fee information, which includes the person's charge information and transaction facts. The auditability, authenticity, availability, and integrity protection annotations ensure that price records are effectively recorded, auditable, and keep their integrity during the transaction.

If the payment permission is successful, the driver is paid using the "Release Payment" procedure. The availability and integrity of safety annotations guarantee that the charge release technique is reliable, available, and free of unauthorized alterations.

Moreover, customers may publish remarks on their experience level by means of finishing the "Provide Feedback" assignment in the "Ride Coordinator and Ride Request" lane. The data item "Feedback" collects consumer input approximately their experience enjoyable. The auditability and authenticity protection annotations make certain that feedback statistics are effectively saved, keep their integrity, and may be tracked and returned to the corresponding consumer.

The "Providing Feedback is Successful" end event is triggered while comments are efficaciously submitted. The auditability and authenticity safety annotations ensure that the feedback submission occasion is reliably documented, authenticates the consumer's input, and preserves its integrity.

The "Payment is Declined (Unsuccessful)" end event is prompted if the charge authorization method fails or is denied. The availability and integrity protection annotations guarantee that the event appropriately records the failing payment authorization and preserves the occasion data integrity.

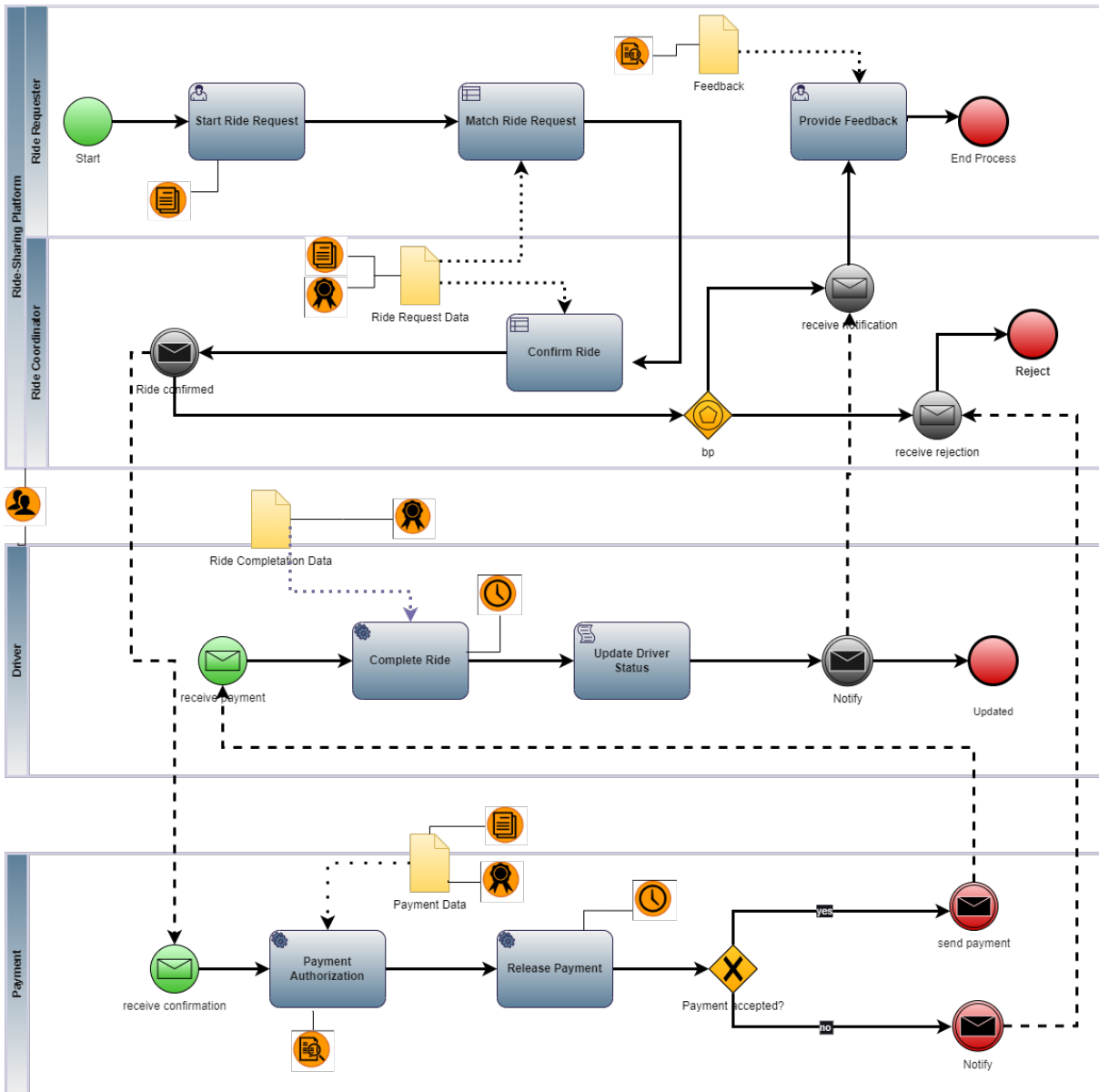


Figure 3.5: Example of SecBPMN2BC diagram for Ride Sharing System

4 | Algorithmic Framework

This chapter introduces a set of rules designed to deal with the demanding situations springing up from conflicting and unsupported security properties in smart contracts within a blockchain framework. The structure and protection capabilities of blockchain necessitate careful consideration of which security properties can be supported on-chain and which have to be accomplished off-chain. Indeed, failure to address those conflicts and boundaries can disclose smart contracts to huge safety vulnerabilities, compromising their integrity and reliability.

Consider the running example of a smart contract process (see Section 3.5) centered on handling the retrieval and authorization of unique statistics from a Ride Sharing System, in such eventualities, conflicting and unsupported security properties can also emerge, especially while multiple protection necessities are assigned to a single element. This state of affairs gives demanding situations in effectively enforcing security features, certainly, conflicts can rise up when attempting to fulfill all assigned security necessities for an element, as sure properties might also contradict or be incompatible with each different.

To tackle this problem, the supplied algorithm aims to discover and optimize the set of security properties that may be completed on-chain, prioritizing the security requirements that make contributions to the very high possible level of safety for the smart agreement. By doing so, the set of rules mitigates the dangers associated with conflicting properties and complements the trustworthiness and protection of smart contracts in the blockchain ecosystem.

When considering the decision to place an element on a blockchain, it is important to observe the advantages it offers, particularly in terms of security [38], as putting an element on-chain can make certain transparency and authenticity, and in the end lower the fee of employing third-party programs to stable the element. These include scalability-demanding situations because the network needs to handle a huge quantity of statistics efficaciously, there can be financial charges related to storing and processing information, consisting of costs for data storage, transaction processing, and network utilization. Moreover, persistence issues can occur due to the immutability of information at the

blockchain, making it hard to modify or delete stored elements [16]. Persistence refers to the capability of the blockchain network to hold the integrity and availability of data stored on-chain over time. It can lead to facts loss or statistics corruption i.e. Errors or inconsistencies in the saved data [4].

Therefore, the algorithms presented in this thesis identify a combination of security properties of SecBPMN2BC for each node in the smart contract, taking into account a global enforcement variable, namely Global Enforcement, that measures the level of security enforcement influenced by its predecessors and ancestors. In doing so, the algorithms can determine whether an element of a smart contract should be executed on-chain or off-chain. The decision-making process for storing elements on-chain in a blockchain network involves considering multiple factors that include the properties of the element being stored, its value, and the desired level of security. By carefully evaluating these aspects, an optimal solution, that enhances the security of the smart contract process and mitigates potential conflicts can be determined. The aim is to find a balance that meets the necessary security requirements while minimizing the risk of conflicts. To test the efficiency of the algorithm, two different algorithms are proposed: a brute force version and an optimized version. The brute force algorithm explores all possible combinations of security properties, while the optimized reduces the number of combinations to explore through a heuristic approach. The subsequent sections will provide a detailed description of the algorithm and its two versions.

4.1. Global Enforcement

It's worth noting that the set of rules for SecBPMN2BC previously discussed in Section 3.4.2 are primarily focused on optimizing security at a local level. To complement this local security analysis, an additional parameter has been introduced: *Global Enforcement*. This parameter is a numerical value given to each set of properties combinations that aims to provide an indication of the overall level of security that is enforced by the blockchain's security properties. Unlike the set of rules which operate on individual nodes, this new parameter calculates the security level based on the relationships between nodes. By considering the security of nodes in relation to one another, this approach is able to provide a global assessment of BC's security. Before delving into the specifics of this variable, it is essential to first explain why it has been implemented.

In more detail, the calculation of global enforcement in the context of combination rules takes into account the local enforcement value of the specific combination. As explained in Section 3.4.2, combinations can have three different levels of local enforcement.

- For native combinations, where the enforcement is inherent and automatic, the global enforcement value is assigned as 1, indicating full enforcement.
- If a combination has no enforcement, meaning there are no rules or measures in place to enforce it, the global enforcement value is assigned as 0, indicating no enforcement.
- In cases where a combination is possible, the global enforcement value is assigned a value between 0 and 1. This value represents the likelihood or probability of the combination being enforced, indeed, the specific value within this range reflects the strength or effectiveness of the enforcement measures associated with that combination.

4.1.1. "Possible" enforcement level

In scenarios where each local property's level of enforcement is possible, we discover distinct outcomes depending on the type of security annotations. It is crucial to recognize that different requirements can offer varying levels of security, and these priorities can be influenced by the stakeholders involved. When local enforcement is "possible" for a particular combination, a value between 0 and 1 can be assigned to represent its level of global enforcement. By assigning such values, it becomes possible to prioritize and compare different requirements based on their respective enforcement capabilities, allowing stakeholders to make informed decisions and allocate resources accordingly. Considering the security requirements described in Section 3.3.1 and the possible rule combinations obtained in Section 3.4.2, we can derive a range of values for the global enforcement in each case.

First and foremost, it is essential to analyze all scenarios where different levels of enforcement can be achieved for each security requirement.

Starting from *Authenticity*, it is possible to achieve possible local enforcement level if `OnChainExecution` is set to true and the `BlockchainType` is public. This signifies that an external mechanism is necessary to validate the user's identity.

For *Auditability*, if `OnChainExecution` is false, the blockchain monitors the node that recognized the beginning of the activity, but it does not assure that the node that sent the notification initiated the activity. In the case of messages, if `OnChainData` is set to false, the blockchain cannot prevent instances where messages are falsified, or notifications of receipt are not delivered. The same problem applies to *Availability*.

Concerning *Integrity*, a possible enforcement level can be attained when *OnChainExecution* is set to false. This means that, in addition to external mechanisms to ensure integrity, all the necessary data for executing the activity must be supplied to the blockchain. If these mechanisms do not operate correctly, the security level will be no enforcement, and the possible value will be closer to no enforcement than native. The same issue arises for *Non-repudiation* (activity). For MFs, if *OnChainData* is set to none, security is only guaranteed for events related to the message, but not for the message's content.

In the case of *NonDelegation*, if *OnChainExecution* is set to false, it implies that the whole process is conducted outside the blockchain. Therefore, the blockchain can solely prevent notifications from unauthorized nodes, leading to a low level of security, thus different values could be allocated to characterize the security level guaranteed for these cases.

When considering the security of elements of a smart contract, whether stored on-chain or off-chain, it is possible to analyze and determine the relative level of security for different elements in each scenario. Based on the analysis of specific cases, a ranking can be established, indicating the elements that are less likely to be secure when exposed to either on-chain or off-chain situations. The ranking ranges from the least secure to the most secure, providing insights into the vulnerabilities and risks associated with different elements in each context.

BOD, *SOD*, and *Non-Delegation* share the commonality that if their enforcement level is achievable, the entire activity will be executed off-chain. As a result, the security level is closer to no enforcement than native, so a low value of global enforcement can be designated.

Auditability, *Availability*, *Integrity*, *Non-repudiation*(activity) share in common that the element is partially protected provided that, in addition to external mechanisms, the required data for executing the element is also made available. Partial security is guaranteed only if all these events occur; otherwise, the security requirement will have "no-enforcement". In contrast to the previous scenarios, the security level, in this case, is higher, indicating the need for a moderate level of global enforcement.

For the remaining properties, i.e., *Authenticity* and *Non-repudiation*(MF), the blockchain secures only specific portions of the element, while the rest is left to external mechanisms. A high global enforcement value could be assigned.

Ultimately, the user has the final decision on the value to set for the global enforcement,

based on their specific needs. When selecting a value for the global enforcement assigned to each node combination of security annotations, it is crucial for the user to consider several factors:

- The user should have a clear understanding of the desired level of security and the potential risks associated with the specific smart contract.
- The user should also take into account the limitations of certain ranges of global enforcement that can be assigned to certain security annotations.
- The user should consider the dependencies and relationships between nodes in the smart contract and how the assigned global enforcement values will propagate through the contract.

By carefully evaluating these factors, the user can make an informed decision when selecting the appropriate global enforcement value for each node in the smart contract.

4.2. Common initial part of the proposed algorithms

In this thesis, two algorithms are proposed: a brute force algorithm and an optimized algorithm. The brute force algorithm involves an implementation of the algorithm without any optimizations. On the other hand, the optimized algorithm involves the application of techniques to make the algorithm more efficient and reduce its computational complexity. The final aim is to compare the results obtained from the two algorithms of the algorithm, in order to verify whether the optimized algorithm that performs better in terms of speed and efficiency produces the same result as the brute force algorithm, which exhaustively explores all possible combinations of security properties in order to identify the best configuration for each node in the smart contract. To evaluate the performance of the two algorithms of the algorithm, I have conducted various experiments and analyzed the results obtained from them.

Before delving into the nuances of the two algorithms of the algorithm, let's first examine the fundamental aspect that they both share: the assignment of security properties to each node. This crucial step forms the foundation upon which the rest of the algorithm is built, as it determines the specific attributes and characteristics that each node will possess. In this initial phase, a set of combinations $S(node)$ is created for each element within the smart contract, representing individual nodes. Each $S(node)$ set consists of all possible combinations of properties that are impacted by the associated security requirements linked to the node, with the local enforcement obtained from Table 3.4, and the

respective global enforcement value. In particular, $S(node)$ consists of the combinations $C_j = [\{P_k\}, le, gle]$ where:

- $P_k = \{property, value\}$ denotes a combination of a property with its value, which can hold for that specific element.
- $le = localEnforcement$ is a categorical variable that can assume one of three possible values: *native*, *possible* or *no_enf*. This variable determines the level of strictness with which the blockchain enforces rule R, with *native* representing the highest level of strictness and *no_enf* representing the lowest level of strictness.
- $gle = globalEnforcement$ is a numerical value that quantifies the degree to which a rule impacts the nodes within a branch in a global sense. Specifically, this score tracks the degree of enforcement of the rule from the leaf to the root of the relevant branch, across all branches of the tree. A higher value indicates greater efficacy of the rule in ensuring node security.

Thus, within the set, each combination consists of distinct security combinations representing various aspects of the node's security requirements. This approach enables the tracking of both parent and child constraints, establishing a comprehensive framework for setting up the entire process. Taking the example of the Confirm Ride task node from the Running Example in Section 3.5 subjected to the *Audiability* security requirement, it acquires a set of property combinations. Each combination comprises three security requirements: *onChainExecution* for the local node, *onChainModel* for its parent, and *blockchainType* for its ancestors. Basically, the *blockchainType* that can be set to public or private belongs to the definition of the process, that is considered the ancestor of the current leaf node, while *onChainModel* is assigned to the Pool in which the leaf node belongs, thus considering its directed parents. Lastly, *onChainExecution* is for the current node, in this specific case it is referred to as a Task. This combination format enables the systematic tracking of parent and child constraints, thereby facilitating effective enforcement and maintaining the integrity of the security measures within the system. In other words, this format guarantee to trace all the security requirement assigned to all the nodes on the same branch of the current one, with the outcome of reducing the risk of conflicts when propagating the security requirements through the process structure. Thus, Figure 4.5 illustrates the set $S_{ConfirmRide}(Authenticity)$:

P_k	LE	GLE
{blockchainType, private}, {onChainModel, true}, {onChainExecution, true}}	native	1
{blockchainType, private}, {onChainModel, true}, {onChainExecution, false}}	no_enf	0
{blockchainType, private}, {onChainModel, false}, {onChainExecution, true}}	native	1
{blockchainType, private}, {onChainModel, false}, {onChainExecution, false}}	no_enf	0
{blockchainType, public}, {onChainModel, true}, {onChainExecution, true}}	native	1
{blockchainType, public}, {onChainModel, true}, {onChainExecution, false}}	native	1
{blockchainType, public}, {onChainModel, false}, {onChainExecution, true}}	no_enf	0
{blockchainType, public}, {onChainModel, false}, {onChainExecution, false}}	no_enf	0

Table 4.1: $S_{ConfirmRide}(Audiability)$

It is crucial to recognize that a single node within the system can be subjected to multiple security requirements, each of which generates its own set of constraints. Consequently, when aiming to establish a coherent and comprehensive set of combinations for a given node, the merging of all individual security requirement sets becomes necessary. However, this merging process can introduce conflicts among the requirements, as different security rules may impose contradictory or incompatible conditions. Conflicts arise when two combinations, namely $C_1 = \langle [P_k]_1, le_1, gle_1 \rangle$ from the rule set $S_{R1}(node)$ (defined in Section 4.2) associated to a specific security requirement and $C_2 = \langle [P_k]_2, le_2, gle_2 \rangle$ from the rule set $S_{R2}(node)$ associated to a different security requirement for the same node,

exhibit the same property pair denoted as $[P_k]_1$ and $[P_k]_2$, while their local enforcement values (le) or global enforcement values (gle) differ ($le_1 \neq le_2$ or $gle_1 \neq gle_2$). This conflict arises due to divergent enforcement values for the same property pair within different sets of rules representing distinct security requirements associated with the same node.

The process involves assessing the local enforcement values (le) for each combination and selecting the combination with the lowest le value, which signifies the least strict constraints so as to emphasize the set of rules that impose fewer restrictions on the specific security requirement. This approach is motivated by the goal of achieving a compromise or resolution when conflicts arise between different security requirements for the same node. It recognizes that these conflicts can impact the overall functionality of the system, while still ensuring that the necessary security requirements are met. Essentially, it aims to address conflicting security requirements by giving precedence to the combination that imposes fewer limitations. This allows for a more adaptable and balanced approach to fulfilling multiple requirements.

The global enforcement value (gle) is updated by calculating the average of the corresponding values from the two sets. This approach aims to reconcile the conflicting enforcement values and find a balanced solution. By taking the average of the gle values, both sets of rules associated with distinct security needs are considered, with the goal of finding a medium ground. This method guarantees that neither set of rules dominates the other and yields a global enforcement value that represents a compromise between the competing ideals. Thus it enables a collaborative approach to dispute resolution and the development of a balanced enforcement plan for the whole system. When multiple security requirements conflict with each other, it can be challenging to determine the most suitable course of action. By selecting the combination with the lowest le value, the approach emphasizes the set of rules that imposes fewer restrictions on the specific security requirement. This means that the chosen combination places fewer limitations on the node's operations or functionalities, allowing it to maintain a higher level of adaptability.

$$gle_2 = \frac{gle_1 + gle_2}{2} \quad (4.1)$$

Thus, in the scenario where $le_1 < le_2$, the combination C_1 is included in the resulting set, along with the updated global enforcement value. Conversely, if $le_1 \geq le_2$, the combination C_2 is added to the resulting set, also with the updated gle value. For the purpose of illustrating the concept, let's consider the Integrity security requirement for the same Confirm Ride task, denoted as $S_{Integrity}(ConfirmRide)$. This task is indirectly linked to the security requirements associated with DataObject Ride completion data. The task

and the object have a connection, which ultimately influences the final composition of the task node-set. In this case, for the sake of simplicity, the global enforcement value is assumed to be equal to 0.5 for the "possible" scenarios, but in a real case it would be parametrized as defined in Section 4.1.1.

Pk	LE	GLE
[{blockchainType, private}, {onChainModel, true}, {onChainExecution, true}]	native	1
[{blockchainType, private}, {onChainModel, true}, {onChainExecution, false}]	possible	0.5
[{blockchainType, private}, {onChainModel, false}, {onChainExecution, true}]	no_enf	0
[{blockchainType, private}, {onChainModel, false}, {onChainExecution, false}]	no_enf	0
[{blockchainType, public}, {onChainModel, true}, {onChainExecution, true}]	native	1
[{blockchainType, public}, {onChainModel, true}, {onChainExecution, false}]	possible	0.5
[{blockchainType, public}, {onChainModel, false}, {onChainExecution, true}]	no_enf	0
[{blockchainType, public}, {onChainModel, false}, {onChainExecution, false}]	no_enf	0

Table 4.2: $S_{ConfirmRide}(Integrity)$

As it is possible to notice between the $S_{ConfirmRide}(Audiability)$ in Table 4.1 and the $S_{ConfirmRide}(Integrity)$ in Table 4.2 there are combinations that run into conflicts.

Pk	LE	GLE
[[{blockchainType, private}, {onChainModel, true}, {onChainExecution, false}]]	no_enf	0
[[{blockchainType, private}, {onChainModel, false}, {onChainExecution, true}]]	native	1
[[{blockchainType, public}, {onChainModel, true}, {onChainExecution, false}]]	native	1

Table 4.3: $S_{ConfirmRide}(Audiablity)$

Pk	LE	GLE
[[{blockchainType, private}, {onChainModel, true}, {onChainExecution, false}]]	possible	0.5
[[{blockchainType, private}, {onChainModel, false}, {onChainExecution, true}]]	no_enf	0
[[{blockchainType, private}, {onChainModel, false}, {onChainExecution, true}]]	possible	0.5

Table 4.4: $S_{ConfirmRide}(Integrity)$

These combinations have the same $\{property, value\}$ but different le and gle values. To ensure, then, the creation of a unique final set for $ConfirmRide$, only the combinations with the lowest local enforcement and global enforcement are selected, following the guidelines outlined in Section 4.2. The final set in Tab 4.5, indeed, $S_{ConfirmRide}(node)$ can be obtained accordingly.

Pk	LE	GLE
[[{blockchainType, private}, {onChainModel, true}, {onChainExecution, true}]]	native	1
[[{blockchainType, private}, {onChainModel, true}, {onChainExecution, false}]]	no_enf	0.25
[[{blockchainType, private}, {onChainModel, false}, {onChainExecution, true}]]	no_enf	0.5
[[{blockchainType, private}, {onChainModel, false}, {onChainExecution, false}]]	no_enf	0
[[{blockchainType, public}, {onChainModel, true}, {onChainExecution, true}]]	native	1
[[{blockchainType, public}, {onChainModel, true}, {onChainExecution, false}]]	possible	0.75
[[{blockchainType, public}, {onChainModel, false}, {onChainExecution, true}]]	no_enf	0
[[{blockchainType, public}, {onChainModel, false}, {onChainExecution, false}]]	no_enf	0

Table 4.5: $S_{ConfirmRide}(node)$

Eventually, the local enforcement mechanisms ensure that the individual security requirements are met, while the global enforcement mechanisms ensure that the overall security of the node is maintained. This approach provides a comprehensive and effective way to manage security requirements for nodes.

4.2.1. Structure of the algorithm

The algorithm consist up of two phases: the *bottom – up* and *top – down* phase. In order to represent the process it is used a tree structure. The root element serves as the

representation for process definitions, allowing for the configuration of the `blockchainType` and `onChainModel` properties. Pools, which act as intermediate elements, are subjected to the `onChainModel` property. Similarly, subprocess activities that are direct children of Pools are considered intermediate elements, so the property `onChainModel` can be set for them. Whereas, as leaf elements, tasks can have `onChainExecution` set, while `onChainData` property can be set for data objects and messages.

More in detail, the objective of the bottom-up phase is to identify every conceivable combination at each node. One of these is picked as the ideal combination and transmitted downward in the second phase to get rid of the incompatible ones determined in the first phase at each node. The optimal combination at each node will ultimately be chosen. Naturally, the two algorithms previously described differences in the criteria for identifying all potential combinations and selecting the best one. The brute-force algorithm will be covered first in the parts that follow, and then a more in-depth discussion of the optimized method will follow.

To gain a deeper understanding of how these two algorithms work, in Section 3.5 there is an example smart contract that serves as a testbed for both methods.

5 | Bruteforce strategy

The brute force approach is a technique used to solve optimization problems by exhaustively evaluating all possible combinations of elements. It involves two stages, namely bottom-up and top-down, ensuring that every potential combination is considered. This algorithm systematically explores each solution without employing optimization techniques to narrow down the search space. It diligently evaluates each combination to identify the optimal solution. Nevertheless, the brute force approach guarantees the discovery of the best solution by exhaustively exploring and evaluating all available combinations. Despite its effectiveness, it is important to note that this approach can be computationally expensive due to the extensive evaluation of a vast number of combinations, particularly for complex problem scenarios.

5.1. Propagate up

The algorithm *propagateUp* employs a brute-force strategy to propagate all possible combinations from leaf nodes to root nodes in a hierarchical process structure. The nodes in this structure are Generalized Markup Tree (GMT) nodes. The algorithm's main goal is to obtain the ultimate set comprising all admissible combinations by iteratively propagating and constraining combinations only when absolutely required. Through this approach, the algorithm exhaustively explores and evaluates potential combinations, ensuring that no viable option is overlooked. In this context, consider a set of combos, denoted $S_{node}(ConfirmRide)$, generated from previous calculations in Table 4.2. These combinations represent different property values assigned for different nodes of the process model across the property value matrices.

Pk	le	gle
[[blockchainType, private], {onChainModel, true}, {onChainExecution, true}]]	native	1
[[blockchainType, private], {onChainModel, true}, {onChainExecution, false}]]	no_enf	0.25
[[blockchainType, private], {onChainModel, false}, {onChainExecution, true}]]	no_enf	0.5
[[blockchainType, private], {onChainModel, false}, {onChainExecution, false}]]	no_enf	0
[[blockchainType, public], {onChainModel, true}, {onChainExecution, true}]]	native	1
[[blockchainType, public], {onChainModel, true}, {onChainExecution, false}]]	possible	0.75
[[blockchainType, public], {onChainModel, false}, {onChainExecution, true}]]	no_enf	0
[[blockchainType, public], {onChainModel, false}, {onChainExecution, false}]]	no_enf	0

Table 5.1: $S_{node}(ConfirmRide)$

Upon closer examination, in Table 5.1 it is observed that each combination is comprised of the following properties:

- *blockchainType*: This property belongs to the ancestor node, specifically the root node.
- *onChainModel*: This property belongs to the parent node, either a Process or a SubProcess (Pool).
- *onChainExecution*: This represents the local property of a leaf node, indicating

the execution of tasks on the blockchain.

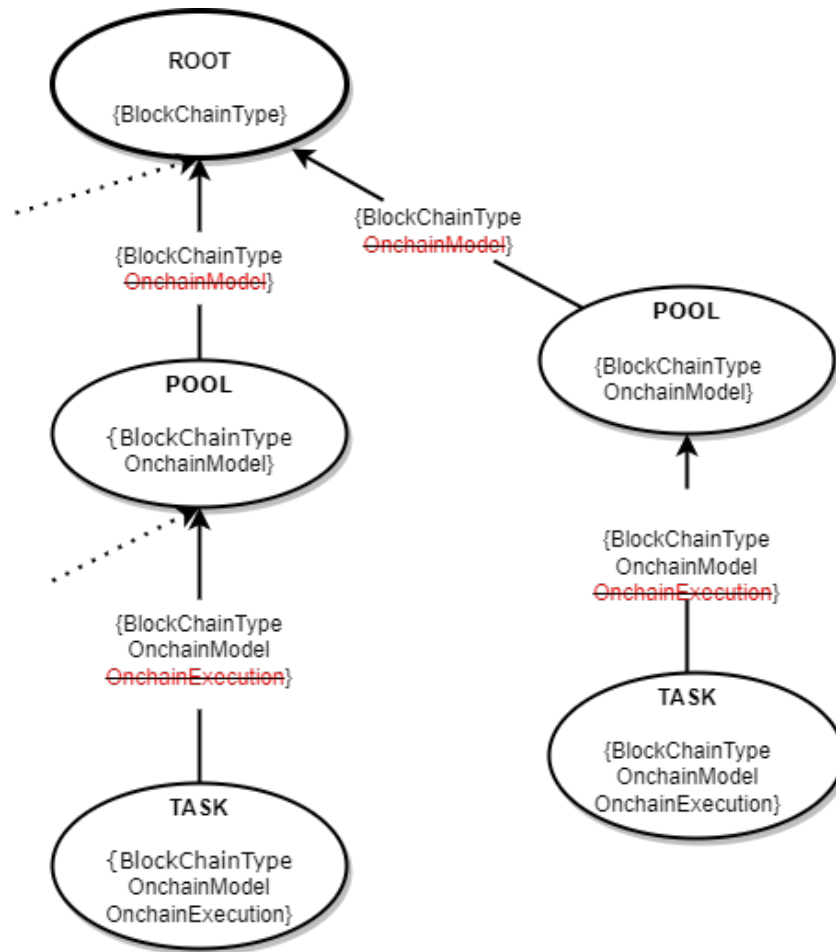


Figure 5.1: Graphical representation of the Propagate-up phases

It is important to note that the set of the parent node will consist only of combinations composed of *blockchainType* and *onChainModel*, as these properties are automatically inherited from the parent node, as well as the root node, which will only include the *blockchainType*. To ensure the effective propagation of constraints from child to parent, it becomes necessary to remove the local properties specific to the current node. This prevents conflicts and allows the parent node to be appropriately constrained. A temporary set denoted $S_{temp}(ConfirmRide)$, is created to achieve this. It comprises all combinations of the current node, excluding its local properties.

Pk	le	gle
[{blockchainType, private}, {onChainModel, true},	native	1
[{blockchainType, private}, {onChainModel, true},	no_enf	0.25
[{blockchainType, private}, {onChainModel, false},	no_enf	0.5
[{blockchainType, private}, {onChainModel, false},	no_enf	0
[{blockchainType, public}, {onChainModel, true},	native	1
[{blockchainType, public}, {onChainModel, true},	possible	0.75
[{blockchainType, public}, {onChainModel, false},	no_enf	0
[{blockchainType, public}, {onChainModel, false},	no_enf	0

Table 5.2: $S_{temp}(ConfirmRide)$

However, during the removal of local properties, a challenge arises. The operation introduces duplicate combinations within the same set, resulting in conflicts in Table 5.2. (for example the 1st and 4th combinations). In fact, $Pk = \text{property, value}$ are the same but their local enforcement value differs. (see Section 4.2), namely $\text{property} = \text{property}' \wedge \text{value} \neq \text{value}'$. Please note that in this case combinations of assignments are removed, not single properties. These rules state that conflicts occur when two combinations have the same property but differ in their corresponding values.

To address this issue and maintain the integrity of the constraints, it is necessary to generate subsets that encompass all possible combinations from $S_{temp}(ConfirmRide)$ while ensuring that no repetition occurs within the same subset. This process leads to the creation of the set S_{up} , which is essentially a list of sets. Each subset within $S_{up}(ConfirmRide)$ is derived by considering all possible cases and combinations.

By splitting the set into subsets without repetitions, the algorithm guarantees that each combination appears only once within $S_{up}(ConfirmRide)$. This step effectively eliminates conflicts arising from duplicate combinations, ensuring the accuracy and validity of the propagated constraints. Due to brevity only four of all possible sets' combinations are

reported in Tables(5.35.55.45.4):

Pk	le	gle
[{blockchainType, private}, {onChainModel, true}]	native	1
[{blockchainType, private}, {onChainModel, false}]	no_enf	0.5
[{blockchainType, public}, {onChainModel, true}]	native	1
[{blockchainType, public}, {onChainModel, false}]	no_enf	0

Table 5.3: $Subset_{up}(ConfirmRide)$

Pk	le	gle
[{blockchainType, private}, {onChainModel, true}]	no_enf	0.25
[{blockchainType, private}, {onChainModel, false}]	no_enf	0.5
[{blockchainType, public}, {onChainModel, true}]	native	1
[{blockchainType, public}, {onChainModel, false}]	no_enf	0

Table 5.4: $Subset_{up}(ConfirmRide)$

Pk	le	gle
[{blockchainType, private}, {onChainModel, true}]	native	1
[{blockchainType, private}, {onChainModel, false}]	no_enf	0
[{blockchainType, public}, {onChainModel, true}]	native	1
[{blockchainType, public}, {onChainModel, false}]	no_enf	0

Table 5.5: $Subset_{up}(ConfirmRide)$

Pk	le	gle
[{blockchainType, private}, {onChainModel, true}]	native	1
[{blockchainType, private}, {onChainModel, false}]	no_enf	0.25
[{blockchainType, public}, {onChainModel, true}]	possible	0.75
[{blockchainType, public}, {onChainModel, false}]	no_enf	0

Table 5.6: $Subset_{up}(ConfirmRide)$

Subsequently, to successfully propagate the constraints from child to parent nodes, the subsets of $S_{up}(ConfirmRide)$ need to be merged with the constraint set of the parent node. This merging process allows the constraints from the child to be incorporated into the parent while considering all possible combinations.

For example the $S_{node}(Driver)$ in Table 5.7) subject to BOD security requirement is the following:

Pk	le	gle
[{blockchainType, private}, {onChainModel, true},	native	1
[{blockchainType, private}, {onChainModel, false},	possible	0
[{blockchainType, public}, {onChainModel, true},	no_enf	0
[{blockchainType, public}, {onChainModel, false},	no_enf	0

Table 5.7: $S_{node}(Driver)$

To accomplish this, the algorithm invokes the *constrainSet* function. This function compares each combination of the subsets with the parent constraint set and adds the combination to the final set, denoted as $S_{final}(Driver)$ shown in Table 5.8 only if it does not create conflicts with the parent's existing constraints. The *constrainSet* function essentially updates the parent constraint set with all the possible combinations obtained from the subsets of $S_{up}(Children)$, In this specific case, for illustrative purposes, only the $S_{up}(ConfirmRide)$ steps are taken into consideration.

Pk	le	gle
[{blockchainType, private}, {onChainModel, true},	no_enf	0.75
[{blockchainType, private}, {onChainModel, false},	no_enf	0.25
[{blockchainType, public}, {onChainModel, true},	possible	0.62
[{blockchainType, public}, {onChainModel, false},	native	1
[{blockchainType, private}, {onChainModel, false},	no_enf	0.30
{...},		

Table 5.8: $S_{final}(Driver)$

The process continues through the entire tree process structure until it reaches the $S_{root}(node)$, along with its corresponding relative $S_{final}(node)$.

5.1.1. Pseudocode Propagate-up Bruteforce

The process involves starting with an input node and initializing an empty set called S_{temp} (line 3). The algorithm then proceeds to iterate through each combination, denoted as C_j , in the set associated with the current node. During this iteration, the local properties specific to that node are removed (lines 5-10). The modified combinations, which now consist of only the parent and ancestor properties, are added to the S_{temp} set (line 9). This allows us to obtain a constrained set that can be compared with the parent node's set, which consists solely of (parent, ancestor) combinations.

If the current node is not a leaf node, indicating the existence of child nodes, the algorithm enters a loop to process each child node (lines 11-32). Within this loop, it initializes three empty sets: S_{parent} to store combinations related to the parent node (line 13), S_{up} to hold combinations propagated up from the child nodes (line 14), and S_{final} to store the final set of admissible combinations (line 15).

For each child node, the algorithm recursively calls the *propagateUp* function to obtain combinations, denoted as S_{up} , which are propagated up from the child nodes (lines 16-17). Within the loop, for each property, P_{up} , in the properties of S_{up} , the algorithm checks if it is not already present in the properties of the current node (lines 18-22). If the property is not present, a new combination, C_{parent} , is created to incorporate this property, and it is added to the set S_{parent} (line 20). This step ensures that the combinations propagated from child nodes are modified to reflect the properties of the current node, eliminating local combinations that are specific to the child nodes.

If the S_{parent} set is not empty, the algorithm adds the split combinations from the S_{parent} set to the S_{up} set (line 26). The split combinations ensure that there are no repeated combinations within each subset, enhancing the integrity of the resulting combinations.

Furthermore, the algorithm checks if the S_{up} set is not empty. If it is not empty, the *constrainSet* function is invoked, taking S_{up} and the temporary set S_{temp} as inputs, and returns the constrained set of combinations (line 28-30). The *constrainSet* function applies further constraints to the combinations, eliminating any conflicting combinations that may arise from the merging process, thereby ensuring a well-defined output.

This iterative process continues as the algorithm progresses from child nodes to parent nodes until it reaches the root node. At each level, the algorithm updates the S_{final} set with the constrained combinations, progressively narrowing down the admissible combinations based on the constraints applied. This comprehensive approach considers all possible combinations while incorporating constraint functions to remove conflicting combinations,

ensuring the reliability and integrity of the resulting code output.

Algorithm 1: propagateUp

Input: SecBPMN2BC node: element**Output:** S_{final} : all possible admissible property value combinations for parent element

```

1   $S_{\text{temp}} = \text{newSet}();$ 
2  for  $C_j$  in  $S_{\text{node}}$  do
3       $C_{\text{temp}} = \text{newCombination}();$ 
4      for  $P_k$  in  $C_j.\text{properties}$  do
5          if  $P_k.\text{name}$  not in  $\text{node.properties}$  then
6               $C_{\text{temp}}.\text{properties.add}(P_k);$ 
7          end
8      end
9       $S_{\text{temp}}.\text{add}(C_{\text{temp}});$ 
10 end
11 if not  $\text{node.isLeaf}$  then
12     for  $\text{child}$  in  $\text{node.children}$  do
13          $S_{\text{parent}} = \text{newSet}();;$ 
14          $S_{\text{up}} = \text{newSet}();;$ 
15          $S_{\text{final}} = \text{newSet}();;$ 
16         for  $C_{\text{child}}$  in  $\text{propagateUp}(\text{child})$  do
17              $C_{\text{parent}} = \text{newCombination}();$ 
18             for  $P_{\text{child}}$  in  $C_{\text{child}}.\text{properties}$  do
19                 if  $P_{\text{child}}$  not in  $\text{node.properties}$  then
20                      $C_{\text{parent}}.\text{properties.add}(P_{\text{child}});$ 
21                 end
22             end
23              $S_{\text{parent}}.\text{add}(C_{\text{parent}});$ 
24         end
25         if  $S_{\text{parent}}$  is not Empty then
26              $S_{\text{up}}.\text{addAll}(\text{splitCombination}(S_{\text{parent}}));$ 
27         end
28         if  $S_{\text{up}}$  is not Empty() then
29              $S_{\text{final}} = \text{constrainSet}(S_{\text{up}}, S_{\text{temp}});$ 
30         end
31     end
32 end
33 return  $S_{\text{final}};$ 

```

The *splitCombination* Function 2 serves the purpose of splitting a given set into all possible combinations, ensuring that no duplicate combinations exist within the resulting subsets. This function is particularly useful for propagating subsets to parent nodes efficiently, avoiding the exclusion of all possible combinations and providing a brute-force approach to the problem.

To achieve this, it begins by initializing an empty set called S_{up} (Line 34), which will store the generated subsets. Additionally, a temporary set named Subset is created to hold each individual subset during the iteration process (Line 35). Moving forward a loop that iterates over each combination, denoted as C_{temp} , in the input set S_{parent} (Line 36). Within this loop, another nested loop is established to iterate over the subsets already present in S_{up} . This enables the algorithm to compare C_{temp} with the existing subsets and determine whether it is already included within any of them. If the combination is not found in the current subset, it is added to the Subset set (Line 37-42). After iterating through all existing subsets, the algorithm checks whether Subset is empty. If it is not empty, it implies that a new subset has been created by including C_{temp} , and therefore, Subset is added to S_{up} (Line 43-45).

Once all iterations are completed, the resulting S_{up} set, containing all possible subsets without duplicate combinations, is returned as the output of the function (Line 47).

Algorithm 2: splitCombination

Input: S_{parent} = all combinations that exclude the local ones for the child element**Output:** S_{up} = a set that contains all possible subsets of S_{parent} , with the condition that no combination occurs more than once within a subset.

```

1  $S_{up} = \text{newSet}\langle\text{Set}\rangle();$ 
2  $S\_sub = \text{newSet}();$ 
3 for  $C_{temp}$  in  $S_{parent}$  do
4    $S\_sub = \text{newSet}();$ 
5   for  $subset$  in  $S_{up}$  do
6     if  $C_{temp}$  not in  $subset$  then
7        $Subset.add(C_{temp});$ 
8     end
9   end
10  if  $Subset$  is not Empty() then
11     $S_{up}.add(Subset);$ 
12  end
13 end
14 return  $S_{up};$ 

```

The *constrainSet* function 3 is designed to constrain a set of combinations by eliminating incompatible combinations and returning an updated set that satisfies specific criteria. It achieves this by comparing properties and values between combinations and updating the global enforcement value when applicable. More in detail, the function takes two input parameters: the list of subsets (S_{up}) and the combination of the parent node (S_{constr}). It aims to identify and retain only those combinations from the subset of S_{up} that meet the constraints imposed by S_{constr} .

The function initializes an empty set called S_{ret} , which will store the constrained property value combinations (Line 2). It then proceeds with a series of nested loops to iterate through each subset in S_{up} and each combination in the subset (Lines 3-19). For each combination C_j in a subset, the function compares it with combinations in S_{constr} (Lines 4-19). It employs nested loops to iterate through each combination C_w in S_{constr} and examines the properties of C_w and C_j (Lines 5-19). Within the innermost loops, the function checks if a property is C_w and has a corresponding property in C_j with the same name and value (Lines 9-12). If a matching property is found, the function sets a boolean variable *satisfied* to true and updates the global enforcement value (*gle*) of the property in C_w by averaging it with the *gle* of the corresponding property in C_j (Line 12). Conversely, if no matching property is found, the function sets *satisfied* to false, indicating that the

current combination C_w does not satisfy the constraints imposed by C_j (Line 14).

After examining all properties in C_w , the function checks the value of *satisfied*. If it remains false, the function determines that at least one property in C_w fails to meet the constraints of C_j . In this case, the function sets the boolean variable *found* to false and breaks out of the innermost loop (Lines 15-16). If *found* is still true after checking all combinations in S_{constr} , it means that C_j satisfies the constraints imposed by at least one combination in S_{constr} . The function adds C_j to the S_{ret} set and breaks out of the loop to proceed to the next subset (Lines 16-18).

The process continues until all combinations in all subsets have been examined. Finally, the function returns the resulting S_{ret} set, which contains the constrained property value combinations that satisfy the specified constraints (Line 19).

Algorithm 3: constrainSet

input : List S_{up} : list of subset , S_{constr} : combination of parent node

output: S_{ret} : constrained property value combinations

```

1   $S_{ret}$ =newSet();
2  for  $S\_sub$  in  $S_{up}$  do
3      for  $C_j$  in  $S\_sub$  do
4          for  $C_w$  in  $S_{constr}$  do
5              found=true;
6              for  $P_l$  in  $C_w.properties$  do
7                  satisfied=false;
8                  for  $P_k$  in  $C_j.properties$  do
9                      if  $P_l.name = P_k.name$  and  $P_l.value = P_k.value$  then
10                         satisfied=true;
11                          $P_l.gle = (P_l.gle + P_k.gle)/2$  ;
12                         break;
13                     if satisfied=false then
14                         found=false;
15                         break;
16                 if found=true then
17                      $S_{ret}.add(C_j)$ ;
18                     break;
19 return  $S_{ret}$ ;

```

5.2. Propagate down

In this phase, the objective is to derive a combination from each set that is potentially the most suitable in terms of security. Once the combination, referred to as the *bestcombination*, has been selected, it will be propagated from the root to the leaf, and each set will be constrained based on this combination. Specifically, each S_{node} will be updated to eliminate combinations that conflict with the best combination passed by the parent node.

As the algorithm used in this phase is a brute force version, it iteratively considers one combination at a time for each set and propagates it down the tree. After evaluating all possible combinations, the best combination is selected based on a criterion that enforces security annotations that are inherently provided by the blockchain itself. Thus, the algorithm prioritizes combinations with higher local enforcement, and among those remaining, it selects the one with the highest global enforcement.

Upon the conclusion of the propagate-up phase, the resulting root set is inevitably influenced by the constraints imposed by its children. Consequently, the root set emerges as a list comprising combinations of only *blockchainType* properties that have been inherited from its children. This can be exemplified through an illustrative output, such as: $S_{final}(root)$:

Pk	le	gle
[{blockchainType, private},	no_enf	0.45
[{blockchainType, private},	no_enf	0.25
[{blockchainType, public},	possible	0.75
[{blockchainType, private},	no_enf	0.25
[{blockchainType, public},	native	1
[{blockchainType, private},	possible	0.55
{...}	{...}	{...}

Table 5.9: $S_{final}(root)$

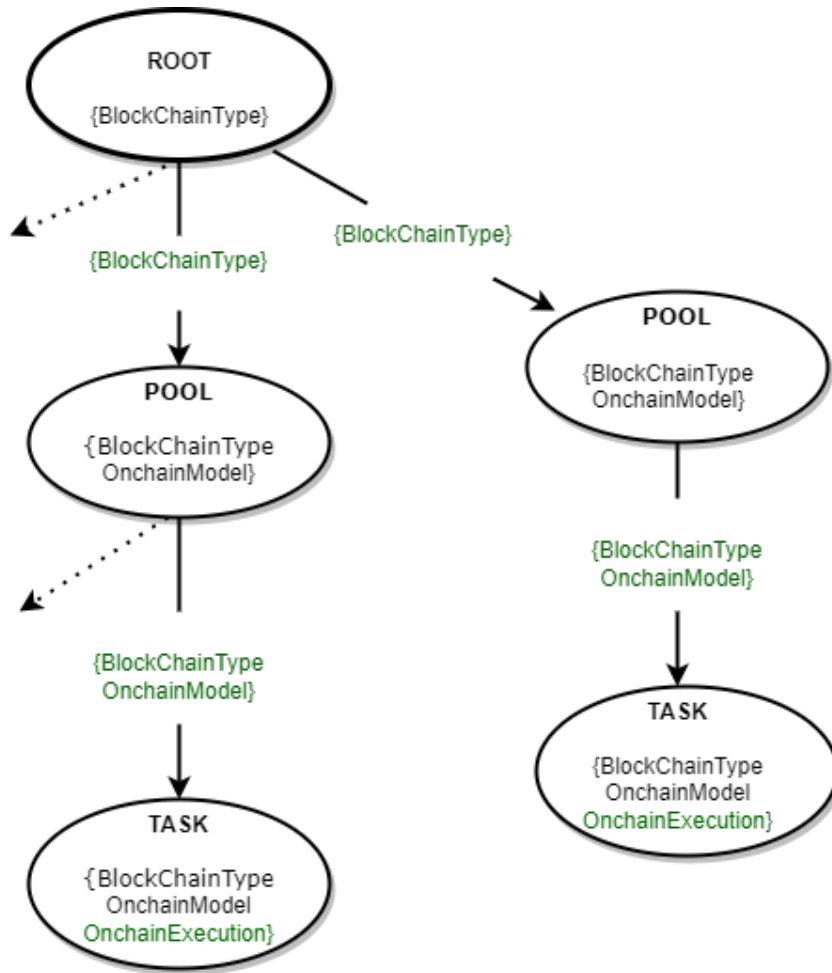


Figure 5.2: Graphical representation of the Propagate-down phases

In Figure 5.2 a simplified root tree is depicted, presenting a graphical representation of the propagate down phases. For the sake of simplicity, only three nodes are considered, and the graph illustrates how combinations are propagated downwards.

To optimize security, the system is performing brute-force on sets and propagating each combination to determine the best fit. The *selectBestCombination* algorithm is utilized for this purpose:

- When a node is not a leaf, the algorithm recursively selects each combination of the previously constrained set $S_{final}(node)$ and designates it as the *bestcombination*. The system then saves this combination and propagates it down to the children nodes by updating the $S_{best}(node)$ set that at each iteration will include the current selected *bestcombination*. This process is repeated until all combinations of the parent set $S_{final}(node)$ have been propagated down .

Next, The algorithm utilizes the *Constraint* function to constrain the best com-

combination of the parent node to its children nodes by imposing the constraint of $S_{best}(node)$ to $S(child)$. This function eliminates any combinations from the set $S(child)$ that are not compatible with the best combination.

- When a node is a leaf, the set $S(child)$ is unconstrained by any child. Therefore, a distinct method is used to determine the optimal combination to be selected in this case. To be specific, the set $S_{Best}(child)$ is derived from $S(child)$ which has already been constrained by the best combination passed down by the parent node, by filtering only those combinations $C_j = [P_k, value]$ where $value = native$. If $S_{Best}(child)$ is empty, then $S_{final}(child)$ is obtained by filtering only those combinations $C_j = [P_k,]$ where $value = possible$ from $S(child)$.
- Furthermore, if $S_{Best}(child)$ is still empty, then $S_{Best}(child)$ is formed by adding only those combinations $C_j = [P_k,]$ where $value = no_enf$. from. In this manner, only the combinations that provide the maximum enforcement are selected. In case the set is still empty after the aforementioned steps, it indicates a conflict in the security requirements, and the process designer must resolve it manually. Eventually, the C_j with the highest value of gle is selected. In the case of a conflict, the designer decides the preferred selection.

To clarify further, let's focus on the first Combination from the set of options called $S_{root}(node)$. This combination is selected by the algorithm and placed in its own separate set, taking the first combination, denoted as $S_{best}(node)_1$, to limit the combination available to the child nodes during the subsequent steps of the process.

Pk	le	gle
{blockchainType, private},	no_enf	0.45

Table 5.10: $S_{best}(root)_1$

In order to propagate it down to the child node, that in this case is $S_{node}(Driver)$, the algorithm invokes the *Constrain* Function 3 in order to properly constrain the combination to the children nodes. As a consequence, the resultant set excludes all combinations with identical $\{P_k\}$ values that have different le values. The set under consideration $S_{node}(Driver)$ 5.8 becomes:

Pk	le	gle
[[blockchainType, private], {onChainModel, true}],	no_enf	0.75
[[blockchainType, private], {onChainModel, false}],	no_enf	0.25
[[blockchainType, public], {onChainModel, true}],	possible	0.62
[[blockchainType, public], {onChainModel, false}],	native	1
{...},		

Table 5.11: $S_{node}(Driver)_1$

It should be noted that combinations 3 and 4 remain intact as the *blockchainType* is different from that of the *BestCombinations*. While they both are *public*, the other is *private*, thereby avoiding any conflict. However, the 5th of the previous set 5.11 combination had both *private* blockchains, but the *blockchainType* of one was *native*, which was clearly distinct from *no_enf*.

Pk	le	gle
[[blockchainType, private],	no_enf	0.45

Table 5.12: $S_{best}(root)_1$

Pk	le	gle
[[blockchainType, private],	native	1

Table 5.13: first combination of $S_{node}(Driver)_{5.11}$

The algorithm systematically evaluates all possible combinations within the set $S_{node}(Driver)$, and in an iterative way, selects the optimal combination until the leaf node is reached.

At this point, the primary objective is to select the combination that can deliver the highest enforcement level, as previously stated. The set $S_{temp}(ConfirmRide)$ is initially constrained based on the best combination of the current iteration obtained from $S_{node}(Driver)_{5.11}$, and subsequently further constrained using the function *getBestCombination* to select only those combinations labeled with the highest local enforcement values le .

Suppose, for instance, that the best combination obtained from $S_{node}(Driver)$ is saved in the set denoted as $S_{Best}(Driver)_1$, and its contents are as follows:

Pk	le	gle
[{blockchainType, private},	no_enf	0.75

Table 5.14: $S_{Best}(Driver)_1$

Pk	le	gle
[{blockchainType, private}, {onChainModel, true}, {onChainExecution, false}]	no_enf	0.25
[{blockchainType, private}, {onChainModel, false}, {onChainExecution, true}]	no_enf	0.5
[{blockchainType, private}, {onChainModel, false}, {onChainExecution, false}]	no_enf	0
[{blockchainType, public}, {onChainModel, true}, {onChainExecution, true}]	native	1
[{blockchainType, public}, {onChainModel, true}, {onChainExecution, false}]	possible	0.75
[{blockchainType, public}, {onChainModel, false}, {onChainExecution, true}]	no_enf	0
[{blockchainType, public}, {onChainModel, false}, {onChainExecution, false}]	no_enf	0

Table 5.15: $S_{best}(ConfirmRide)_1$

Following the proposed strategy, the set will undergo modifications through a selection process aimed at identifying the combination with the highest local enforcement le and global enforcement gle values. Specifically, the selection process will involve identifying the combination with the highest le value, followed by the one with the highest gle value.

It is worth noting that, in this case, the selection process will yield a single combination as the highest le and gle values coincide, therefore the selected combination is uniquely determined. Consequently, the combination with the highest le and gle values will be the sole element of the modified set:

Pk	le	gle
{blockchainType, public}, {onChainModel, true}, {onChainExecution, true}}	native	1

Table 5.16: $S_{best}(ConfirmRide)$

5.2.1. Pseudocode PropagateDown Bruteforce

The *propagateDown* algorithm aims to propagate a combination from the current set down to the leaf nodes in the SecBPMN2BC process tree. It operates in a brute-force manner, considering each combination of the root set as the best combination and propagating it down. It starts by taking a SecBPMN2BC node as input. If the node has a parent (line 1), the algorithm constrains the current set by combining the best combination from the parent set with the current set (line 3). This ensures that the combinations propagate down consistently. Next, the algorithm iterates through each combination, denoted as C_j , in the set associated with the current node (S_{node}). It selects C_j as the best combination (C_{best}) (line 4). To keep track of the best combination, a new set called S_{best} is created to store it (line 5).

The algorithm then recursively calls itself for each child of the current node (line 7). This recursive step ensures that the propagation continues down the tree, exploring all possible combinations. When a leaf node is reached (line 8), it means that the branch in the tree has come to an end. At this point, the algorithm constrains the current leaf set by combining it with the parent's set (line 9). In particular, it constrains the leaf node with the current best combination of the parent set. This step ensures that the combination at the leaf node is consistent with the parent combination.

Finally, the algorithm calls the `getBestCombination` function (line 10) to select the best combination for the leaf node. This function analyzes the combinations in S_{node} and returns the combination providing the maximum enforcement. The algorithm creates a new set, S_{node} , containing only the best combination (line 11) and updates the current leaf set. This process continues until all nodes in the SecBPMN2BC process tree have been traversed

Algorithm 4: propagateDown

input : SecBPMN2BC node: node

```

1 if node.parent then
2    $S_{node} = \text{constrain}(S_{best}, S[\text{node.parent}]);$ 
3   for  $C_j$  in  $S_{node}$  do
4      $C_{best} = C_j;$ 
5      $S_{best} = \text{newSet}(C_{best});$ 
6   for child in node.children do
7     propagateDown(child);
8 if node is leaf then
9    $S_{node} = \text{constrain}(S_{node}, S[\text{node.parent}]);$ 
10   $C_{best} = \text{getBestCombination}(\text{node});$ 
11   $S_{node} = \text{newSet}(C_{best});$ 

```

The *getBestCombination* in Function 4 aims to identify the combination that provides the maximum enforcement within a SecBPMN2BC node. The algorithm begins by calling the *getMaxgle* function with the label *native* to obtain a set of combinations, S_{final} , that satisfy the *native* enforcement criterion (line 1). If the size of the set is zero, indicating no combinations meet the *native* local enforcement, the algorithm proceeds to invoke the *getMaxgle* function again with the label *possible* to obtain a new S_{final} (lines 2-4). If S_{final} is still empty, the algorithm invokes the *getMaxgle* function once more with the label *no_enf*. to obtain the final S_{final} set (lines 6-8). If S_{final} remains empty after these checks, indicating a conflict, an error is raised (lines 9).

The *getMaxgle* Function 6, on the other hand, takes a SecBPMN2BC node and a label representing the desired local enforcement as inputs. It initializes a combination called maxgle (line 1). The function iterates over each combination, C_j , within the node's set of combinations, S_{final} (line 2). For each combination, it checks if the local enforcement, $C_j.le$, matches the provided label (line 74). If a match is found, it compares the global enforcement value of the current combination, $C_j.gle$, with the global enforcement value of the current maxgle combination. If maxgle is empty or $C_j.gle$ is greater than maxgle.gle, maxgle is updated to the current combination (lines 3-5). After processing all combinations, the function returns the combination with the highest global enforcement value(line 6).

The collaborative utilization of the *getMaxgle* and *getBestCombination* functions facilitates the selection of combinations that satisfy the specified local enforcement constraints

while prioritizing those with higher global enforcement values. This approach ensures the identification of combinations that provide strong security measures, emphasizing the importance of selecting combinations that enhance the security of the SecBPMN2BC node when interacting with the blockchain.

Algorithm 5: getBestCombination

input : SecBPMN2BC node: node

output: Combination C_{best} : combination providing maximum enforcement

```

1  $S_{final}$ =getMaxgle(node,'native');
2 if  $S_{final}.size=0$  then
3   |  $S_{final}$ =getMaxgle(node,'possible');
4 if  $S_{final}.size=0$  then
5   |  $S_{final}$ =getMaxgle(node,'no_enf.');
```

```

6 if  $S_{final}.size=0$  then
7   | err.raise(e_i,'Conflict detected');
8 else
9   | return  $S_{final}.get(0)$ 
```

Algorithm 6: getMaxgle

input : SecBPMN2BC node: node, String label: strength

output: Set S_{final} : property value combinations with specified strength

```

1 maxgle = newCombination();
2 for  $C_j$  in  $S_{node}$  do
3   | if  $C_j.le=label$  then
4     | | if maxgle is Empty //  $C_j.gle > maxgle.gle$  then
5       | | | maxgle =  $C_j$ 
```

```

6 return maxgle;
```

6 | Optimized strategy

The optimized approach solves optimization problems and is a more efficient algorithm that aims to find the best combination of elements. Instead of exhaustively considering all potential combinations, this approach carefully considers combinations that meet the desired criteria while disregarding those that are not optimal. Selectively focusing on the most promising combinations efficiently eliminates unnecessary computations and streamlines the search for the best solution. Eventually, this approach leads to faster convergence, reduced search space, and improved overall efficiency. As mentioned earlier in Section 4.2, this algorithm consists of two distinct phases: a bottom-up phase and a top-down phase. The following paragraph provides a detailed description of each phase.

6.1. Propagate up

The `propagateUp` algorithm is designed to facilitate the propagation of property constraints from child nodes to their parent nodes within a tree structure. Given a node representing a Generalized Markup Tree (GMT) node, the algorithm aims to determine the set S_{final} containing all the feasible combinations of property values that can be applied to the parent node. Unlike a brute-force approach that considers all possible combinations, this algorithm selectively considers combinations that are compatible within the sets.

By taking into consideration the same set, from Section 3.5, shown in Figure 6.1 :

Pk	le	gle
[[{blockchainType, private}, {onChainModel, true}, {onChainExecution, true}]]	native	1
[[{blockchainType, private}, {onChainModel, true}, {onChainExecution, false}]]	no_enf	0.25
[[{blockchainType, private}, {onChainModel, false}, {onChainExecution, true}]]	no_enf	0.5
[[{blockchainType, private}, {onChainModel, false}, {onChainExecution, false}]]	no_enf	0
[[{blockchainType, public}, {onChainModel, true}, {onChainExecution, true}]]	native	1
[[{blockchainType, public}, {onChainModel, true}, {onChainExecution, false}]]	possible	0.75
[[{blockchainType, public}, {onChainModel, false}, {onChainExecution, true}]]	no_enf	0
[[{blockchainType, public}, {onChainModel, false}, {onChainExecution, false}]]	no_enf	0

Table 6.1: $S_{node}(ConfrimRide)$

As described in the brute-force algorithm, in order to propagate child constraints to parent nodes, the set S_{temp} Figure 6.2 needs to be transformed into a set that eliminates the local property. Following the previous procedure, the new set obtained is:

Pk	le	gle
$\{\{\text{blockchainType, private}\},$ $\{\text{onChainModel, true}\},$	native	1
$\{\{\text{blockchainType, private}\},$ $\{\text{onChainModel, true}\},$	no_enf	0.25
$\{\{\text{blockchainType, private}\},$ $\{\text{onChainModel, false}\},$	no_enf	0.5
$\{\{\text{blockchainType, private}\},$ $\{\text{onChainModel, false}\},$	no_enf	0
$\{\{\text{blockchainType, public}\},$ $\{\text{onChainModel, true}\},$	native	1
$\{\{\text{blockchainType, public}\},$ $\{\text{onChainModel, true}\},$	possible	0.75
$\{\{\text{blockchainType, public}\},$ $\{\text{onChainModel, false}\},$	no_enf	0
$\{\{\text{blockchainType, public}\},$ $\{\text{onChainModel, false}\},$	no_enf	0

Table 6.2: $S_{temp}(ConfrimRide)$

It is important to note that in this alternative algorithm of the algorithm, the management of conflicting combinations within the same set is approached differently compared to the brute force algorithm. In the previous algorithm, all possible sets were derived, and no occurrences of the same combination were allowed within a single set. However, in the current algorithm, the constraint is applied only within the $S_{node}(temp)$ and the parent node set in this specific case is $S_{node}(parent)$. Consequently, not all combinations are considered, but only those that do not create conflicts. The approach for selecting the appropriate combination follows the guidelines described in Section 4.2. Specifically, the combination with the same property value as the other combination is chosen, but with a lower local global enforcement. Subsequently, the global enforcement is updated as the mathematical average between the global enforcement values of the two combinations from the two different sets. This process results in a constrained parent set node that satisfies the security requirements imposed by its child nodes. To clarify this with an example, considering the same case as the previous algorithm, the parent node set $S_{node}(Driver)$ in Figure 6.3 is now constrained.

Pk	le	gle
[{blockchainType, private}, {onChainModel, true},	native	1
[{blockchainType, private}, {onChainModel, false},	possible	0
[{blockchainType, public}, {onChainModel, true},	no_enf	0
[{blockchainType, public}, {onChainModel, false},	no_enf	0

Table 6.3: $S_{node}(Driver)$

As a consequence of constraining both the $S_{temp}(ConfrimRide)$ and $S_{node}(Driver)$ node sets, the resulting set comprises merely four combinations. Upon comparing the first combination of the $S_{temp}(ConfrimRide)$ set with the first combination of the S_{bod} set, it becomes apparent that they share identical Pk : $\langle \text{property, value} \rangle$, thereby signifying the absence of conflicts. Similar circumstances arise when examining the second combination of the $S_{temp}(ConfrimRide)$ set and the first combination of the parent node set, further confirming the absence of conflicts. To determine the optimal combination, a selection process is employed, wherein the combination with the lowest local enforcement value and reduced global enforcement is chosen. Consequently, the second combination is retained in the final set for the parent node in Figure 6.4, resulting in the following configuration:

Pk	le	gle
[{blockchainType, private}, {onChainModel, true},	no_enf	0.62
[{blockchainType, private}, {onChainModel, false},	possible	0.25
[{blockchainType, public}, {onChainModel, true},	no_enf	0.37
[{blockchainType, public}, {onChainModel, false},	no_enf	0

Table 6.4: $S_{final}(Driver)$

The described process is repeated until the definition node ($S_{root}(node)$) is reached. This iterative procedure aims to constrain each child combination according to the child con-

straints within the entire process structure. By deriving the set $S_{temp}(Driver)$ from the previous step, the local property is eliminated, ensuring that the child constraints propagate effectively to the parent nodes. This ensures that each child combination adheres to its corresponding child constraint throughout the entire process structure.

6.1.1. Pseudocode Propagate-up Optimized

To initiate the process, in Algorithm 7 the algorithm creates an empty set called S_{temp} , which serves as a temporary storage for combinations of properties (line 1). This set includes all the combinations from the node's set and removes the local property, allowing the remaining properties to be propagated to the parent node.

In order to do what is described, the algorithm iterates over each combination C_j in the set $S(\text{node})$. Here, $S(\text{node})$ represents the existing combinations of properties for the current node (lines 2-9). For every C_j , a new empty combination C_{temp} is created to hold the modified properties (line 3). The algorithm proceeds to examine the individual properties P_k within C_j and checks if the property name is absent in the properties of the node. If the condition holds true, indicating that the property is not specific to the node but inherited from its ancestors, P_k is added to the C_{temp} combination (lines 6-8). Finally, the updated C_{temp} combination is appended to the S_{temp} set (line 9). This step ensures that each combination in S_{temp} only contains properties that are not specific to the current node but rather inherited from higher-level nodes. As a result, S_{temp} represents the possible property combinations that can be applied to the parent node. Note that the algorithm does not modify the original $S(\text{node})$ set; it only creates a temporary set S_{temp} with modified combinations.

If the current node is not a leaf node, the algorithm propagates constraints from the child nodes to the parent node (lines 11-28). It initializes an empty set S_{parent} to store combinations belonging to the parent node (line 12).

The algorithm continues by iterating over each child node of the current non-leaf node. For each child node encountered, the algorithm recursively invokes the `propagateUp` function on that child node. The result of this recursive call is stored in a set S_{up} (line 14). This recursive call to `propagateUp` on the child node allows the algorithm to propagate the property constraints from the child node toward its parent node, thus ensuring that property constraints are effectively propagated upwards through the tree structure. The algorithm proceeds to iterate over each combination C_{child} in the set S_{up} (lines 15-23). For each C_{child} , a new empty combination C_{parent} is created to hold the modified properties (line 27). The algorithm then examines the individual properties P_{child} within C_{child} and

checks if the property is not already present in the properties of the current node. If the condition is satisfied, indicating that the property is not specific to the current node but rather inherited from its child node, thus, the property P_{child} is added to the C_{parent} combination (lines 29-33). Finally, the updated C_{parent} combination is added to the S_{parent} set (line 22).

If S_{up} is not empty, the algorithm constraints S_{up} with S_{final} using the constrain function, which applies additional constraints based on the parent's properties and updates S_{final} accordingly (lines 24-26). Finally, the algorithm returns S_{final} , representing all possible admissible combinations of property values for the parent node (line 29).

This algorithm ensures that property constraints are propagated from child nodes to their parent nodes, allowing for consistent and constrained combinations of property values throughout the GMT structure.

Algorithm 7: propagateUp

Input: node: GMTNode**Output:** S_{final} = all possible admissible property value combinations for parent
node

```

1   $S_{temp}$  = newSet();
2  for  $C_j$  in  $S(node)$  do
3       $C_{temp}$  = newCombination();
4      for  $P_k$  in  $C_j.properties$  do
5          if  $P_k.name$  not in  $node.properties$  then
6               $C_{temp}.properties.add(P_k)$ ;
7          end
8      end
9       $S_{temp}.add(C_{temp})$ ;
10 end
11 if not  $node.isLeaf$  then
12      $S_{parent}$  = newSet();
13     for  $child$  in  $node.children$  do
14          $S_{up}$  = propagateUp( $child$ );
15         for  $C_{child}$  in  $S_{up}$  do
16              $C_{parent}$  = newCombination();
17             for  $P_{child}$  in  $C_{child}.properties$  do
18                 if  $P_{child}$  not in  $node.properties$  then
19                      $C_{parent}.properties.add(P_{child})$ ;
20                 end
21             end
22              $S_{parent}.add(C_{parent})$ ;
23         end
24         if  $S_{up}$  is not empty then
25              $S_{final}$  = constrain( $S_{up}$ ,  $S_{temp}$ );
26         end
27     end
28 end
29 return  $S_{final}$ ;

```

The *Constraint* Function 8 plays a crucial role in comparing combinations between the child and parent sets, aiming to handle conflicts and update the parent's set accordingly. By carefully evaluating the compatibility of combinations, it ensures that only compliant

combinations are retained in the parent's set, preserving an adherence to the specified constraints outlined in Section 4.2.

To start, the algorithm initializes an empty set called S_{ret} , which will store the satisfying combinations (line 1). It then iterates through both the input set, S_{input} , and the constraint set, S_{constr} , to analyze their combinations (lines 2-3).

During this nested iteration, the algorithm sets the *found* boolean variable to true, signifying the expectation of finding a satisfying combination (line 33). It then focuses on the properties of the current constraint combination from the second set, initializing the *satisfied* boolean variable as false, indicating that no property has been fulfilled yet (line 6).

By comparing the properties of the input combination with those of the constraint combination, the algorithm seeks matches (lines 7-8). If a match is detected, it proceeds with additional operations, such as updating the global enforcement value of the input combination using the average value derived from both combinations (lines 10-12). Eventually, the function checks the *satisfied* variable to determine if the input combination meets the constraints. If false, the *gle* value is not updated. If true, the combination is updated and added to S_{ret} (line 20), concluding the loop (lines 21-25).

Algorithm 8: CONSTRAINT

Input: S_{input} : set of input combinations, S_{constr} : set of constraint combinations**Output:** S_{ret} : set of combinations satisfying the constraints

```

1  $S_{ret} = \text{newSet}()$ ;
2 foreach  $C_i$  in  $S_{input}$  do
3   foreach  $C_w$  in  $S_{constr}$  do
4     found = true;
5     foreach  $P_I$  in  $C_w.properties$  do
6       satisfied = false;
7       foreach  $P_k$  in  $C_i.properties$  do
8         if  $P_I.name = P_k.name$  and  $P_I.value = P_k.value$  then
9           satisfied = true;
10           $P_I.gle = (P_I.gle + P_k.gle)/2$  ;
11          break;
12        end
13      end
14      if satisfied = false then
15        found = false;
16        break;
17      end
18    end
19    if found = true then
20       $S_{ret}.add(C_i)$ ;
21      break;
22    end
23  end
24 end
25 return  $S_{ret}$ ;

```

6.2. Propagate Down

In this optimized algorithm 9, the approach for selecting the best combination differs from the brute force method. Instead of considering all combinations from the set and saving them as *BestCombination*, the focus is on identifying the most suitable combination based on higher local and global enforcement. This selection process is akin to the step in the brute force algorithm where the best combination for leaf nodes is chosen. During this

phase, the objective is to derive a combination from each set that offers optimal security. Once this *BestCombination* is determined, it is propagated from the root to the leaf, and each set is constrained accordingly. This involves updating each $S(\text{node})$ to eliminate combinations that conflict with the best combination received from the parent node.

Unlike the brute force algorithm, which iteratively considers one combination at a time for each set, the algorithm now prioritizes combinations based on their inherent security annotations provided by the blockchain. After evaluating all possible combinations, the best combination is selected, giving priority to those with higher local enforcement. From the remaining combinations, the one with the highest global enforcement is chosen.

Following the propagate-up phase, the resulting root set is influenced by the constraints imposed by its children. Consequently, the root set consists solely of combinations of *blockchainType* properties inherited from its children. A possible $S_{final}(\text{root})$:

Pk	le	gle
[{blockchainType, private},	no_enf	0.62
[{blockchainType, private},	possible	0.85
[{blockchainType, public},	possible	0.75
[{blockchainType, private},	no_enf	0.12
[{blockchainType, public},	native	1
[{blockchainType, private},	possible	0.25
{...}	{...}	{...}

Table 6.5: $S_{final}(\text{root})$

The method seeks to find the optimal combination from the collection of obtained possibilities. In order to do so, each combination is evaluated in light of the local enforcement requirements. Priority is given to combinations with better local enforcement since they exhibit a stronger adherence to the established security criteria.

Once the local enforcement evaluations are done, the algorithm proceeds to select the combination with the greatest global enforcement from the remaining alternatives. This choice assures that the chosen combination not only meets the local enforcement standards but also delivers the maximum degree of overall enforcement when the larger security environment is taken into account. Thus, in this case the $S_{Best}(\text{root})$ is

Pk	le	gle
[{blockchainType, public},	native	1

Table 6.6: $S_{Best}(root)$

Once the Best Combination is determined, it can be propagated down to the child node, constraining it based on the best combination of the parent set. In the given example, this propagation is illustrated by constraining $S_{Best}(root)$ and $S_{final}(Driver)$ to obtain $S_{Best}(Driver)$. This constraint is applied while adhering to the conflict rules specified in 4.2. The resulting $S_{Best}(Driver)$ combination set is obtained:

Pk	le	gle
[{blockchainType, private}, {onChainModel, true},	no_enf	0.75
[{blockchainType, private}, {onChainModel, false},	no_enf	0.25
[{blockchainType, public}, {onChainModel, false},	native	1

Table 6.7: $S_{best}(Driver)$

Upon careful observation, it becomes evident that only three combinations exist in the set $S_{best}(Driver)$. Notably, the third combination was removed due to a conflict with the best combination passed by the parent node, $S_{Best}(root)$, as they both contained [blockchainType, public]. Although these combinations differed in their local enforcement values (with the former being *possible* and the latter being *native*) to ensure security, it is necessary to eliminate the combination with the lowest local enforcement value. This step is crucial in prioritizing combinations that exhibit stronger adherence to the specified local enforcement criteria.

Moving forward, the aforementioned process is repeated for the obtained set, selecting the best combination as previously described. So in this specific case the Best Combination for $S_{best}(Driver)$

Pk	le	gle
[{blockchainType, private}, {onChainModel, false},	no_enf	0.25

Table 6.8: Best Combination for $S_{best}(Driver)$

This iterative process continues until the leaf nodes are reached, ultimately resulting in the identification of the best combination, speaking in security terms, for each node.

By determining the best combination for each node, security considerations are prioritized, ensuring that operations are carried out directly on the blockchain. This approach minimizes potential vulnerabilities that could arise from off-chain execution or reliance on external systems. Emphasizing on-chain execution enhances the overall security of the system by leveraging the inherent security features and decentralized nature of the blockchain.

6.2.1. Pseudocode PropagateDown Optimized

The *PropagateDown* Algorithm 9, depicted in lines 1-10, aims to identify the best combination for each set and propagate it down the tree structure while constraining the child node sets. The process involves iteratively selecting the best combination from the parent set, updating the parent set with only this best combination, and subsequently constraining the child set based on the updated parent set. More in detail, it accepts a current element, referred to as *node*, as its input (line 2), and is responsible for disseminating the associated constraints to its child nodes through the *Constrain* function, already described in Algorithm 8. The algorithm dedicates lines 3 to 9 to process each combination within the set $S(node)$. During this iteration, it determines the optimal combination by invoking the *getBestCombination* function 5 with the current element, *node*, as an argument (line 3). The resulting combination is stored in the variable C_{best} . The resulting optimal combination is then utilized to construct the set S_{best} (line 5). To enable further constraint propagation within the tree, the algorithm employs recursion by invoking the *propagateDown* function on each child element, denoted as *child* of the current node (line 7). This process ensures that the best combinations propagate down the tree structure, satisfying the constraints at each level.

Algorithm 9: Propagate Down

Input: node - current element**Output:** Propagate constraints down the tree

```

1 if node.parent then
2   |  $S(\textit{node}) = \textit{constraint}(S_{\textit{node}}, S_{\textit{temp}});$ 
3   | for  $C_j$  in  $S(\textit{node})$  do
4     |    $C_{\textit{best}} = \textit{getBestCombination}(\textit{node});$ 
5     |    $S_{\textit{best}} = \textit{newSet}(C_{\textit{best}});$ 
6     |   for each child in node.children do
7       |      $\textit{propagateDown}(\textit{child});$ 
8     |   end
9   | end
10 end

```

This concluding section resembles the approach employed in the Bruteforce algorithm discussed in section 11, As both methods share identical pseudocode, it is unnecessary to reproduce it here. In essence, the *getBestCombination* Function plays a pivotal role in determining the optimal enforcement within a SecBPMN2BC node, follows the same procedure as the previous algorithm of the algorithm, the brute-force algorithm, when selecting the best combination for the leaf nodes in Section 4.

Similar to the brute-force algorithm, the algorithm starts by calling the *getMaxGle* function 4to obtain a set of combinations, $S_{\textit{final}}$, that satisfy the desired local enforcement criterion, initially with the label *native* . If $S_{\textit{final}}$ is empty (indicating no combinations meet the *native* local enforcement), the algorithm proceeds to invoke the *getMaxGle* function again with the label *possible* to obtain a new $S_{\textit{final}}$. If $S_{\textit{final}}$ is still empty, the algorithm invokes the *getMaxGle* function once more with the label *no_enf* to obtain the final $S_{\textit{final}}$ set. If $S_{\textit{final}}$ remains empty after these checks, indicating a conflict, an error is raised.

By utilizing the *getMaxGle* function in a similar manner as the brute-force algorithm, the *getBestCombination* function ensures the selection of the best combination for the leaf nodes while considering the specific local enforcement criteria and prioritizing higher global enforcement values. This approach enhances the security of the SecBPMN2BC node by identifying combinations that provide strong security measures during interactions with the blockchain.

7 | Validation

This section provides the validation and comparison of the two versions of the method (the brute force algorithm and the optimized version). The focus will be on analyzing their performance and providing an accurate analysis of the results.

7.1. Experimental Setup

The experimental evaluation was conducted using SecBPMN2BC modeling language, crafted to support the design of secure business processes to be executed in a blockchain environment. The evaluation was performed within the Eclipse development environment, utilizing the SecBPMN2BC software tool. This tool is based on the STS-Tool libraries and assists in identifying security and privacy conflicts and generating blockchain-related properties for SecBPMN2 elements.

Both the brute force and the optimized algorithms were built and run in the same environment (Eclipse IDE) and tested on the same set of smart contracts, allowing for a direct comparison of their performance. The examinations included were conducted using representative instances of the SecBPMN2BC process, in a variety of situations, beginning with simple examples and proceeding to more difficult ones. These scenarios were created to evaluate the algorithms' performance at various degrees of complexity and to imitate various phases of the SecBPMN2BC process.

In more detail, in order to validate the two algorithms, they had been tested with realistic smart contracts, which permits an evaluation of their performance and effectiveness in real-international conditions. After that, additional exams on both cases were achieved. In particular, corner cases discuss intense or boundary conditions that won't be encountered often but will have a big impact on the conduct and performance of the algorithms. Testing the algorithms in both cases enables the discovery of any weaknesses or vulnerabilities that might not have been obvious all through the sensible situation testing. The following listing represents the scenario analyzed:

- Realistic cases. In order to assess their viability in real-world scenarios, three smart

contract processes were utilized for this purpose. These include the running example discussed in Section 3.5, as well as the hospital teleconsultation process and the road misconstruction process detailed in Paper [18]

- **Boundary conditions.** These are situations in which the input values or parameters of a smart contract attain the acute limits described with the aid of the contract. For example, suppose a process consists of a limit at the most variety of members. In this case, a nook scenario has been exploited in which the agreement is examined with the very best permissible wide variety of participants, as this is aimed at verifying if the contract handles such eventualities appropriately.
- **Exceptional scenarios.** These are situations in which unexpected activities or errors occur inside the smart contract execution. For example, a corner case will be trying out the contract's reaction while encountering factors that do not have particular protection necessities assigned.
- **Unusual inputs.** This entails offering unconventional inputs to the smart contract that might not be encountered in ordinary usage. This can include, unconventional go-with-the-flow sequences, or unusual parameter mixtures.
- **Security conflict vulnerabilities.** Corner cases can also involve testing the contract for potential security conflict vulnerabilities. One approach is to test the contract by imposing additional security requirements on the same objects and observing how the algorithm handles conflicts that may arise

7.2. Evaluated Properties

When comparing two algorithms, it is necessary to consider several criteria to determine which algorithm performs better across different scenarios. To assess the performance of both the optimized and brute-force versions, the following properties have been considered:

- **Execution Time.** It calculates the time the algorithm needs to complete a task and shows how effective it is at processing data quickly.
- **Memory Usage.** Measures the amount of memory an algorithm uses in running time. It represents a crucial factor to take into account because it has an impact on the algorithm's scalability and resource needs.
- **Accuracy.** Shows how close the algorithm's output of the optimized algorithm is to the correct or expected output of the brute force approach.
- **Robustness.** It measures the capacity to effectively manage unexpected or incorrect

inputs, and in handling corner cases, edge cases, and exceptional scenarios without crashing or producing incorrect results.

- **Scalability.** Measures how well an algorithm can handle larger input sizes while maintaining acceptable performance and assesses despite the size increment of the algorithm

7.3. Results Analysis

Realistic scenarios

The brute-force model, via exploration of all possible combos, ensures an increase in the likelihood of finding an optimal solution. However, this exhaustive seek strategy isn't optimal in phrases of time and reminiscence consumption, as it examines every single aggregate inside the process tree. The optimized model, in comparison, is quicker and consumes less reminiscence, permitting it to gain the preferred final results the answer obtained aligns with the one deemed the quality in the brute-pressure approach. In phrases of accuracy, it remains constant, while in phrases of scalability, it surpasses the brute-force technique. Through the implementation of optimization techniques, results are achieved more effectively and with reduced computational overhead.

Boundary conditions

To push the limits and evaluate the algorithm's performance, I conducted experiments by exponentially growing the wide variety of elements concerned. For instance, I tested the algorithm's conduct by assigning a sizable wide variety of tasks and records gadgets to an unconnected pool, without necessarily incorporating new safety requirements. By subjecting the algorithm to such extreme situations, I aimed to assess its robustness and scalability. These experiments allowed me to look at how the algorithm treated the increasing workload and the way it tailored to the growing quantity of factors involved. As expected, the exponential increase in the range of factors may additionally have made the brute-force technique much less powerful as the dimensions of the issue will increase due to its exhaustive nature, the brute force approach turns into extra computationally in depth. As a result, its performance may additionally degrade appreciably, making it less suitable for dealing with larger-scale instances. On the opposite hand, the optimized version of the set of rules probably demonstrated better in keeping with performance under those situations. By using optimization techniques this targeted method reduces the computational complexity and makes the set of rules more scalable. Consequently, the optimized model is lighter in terms of computational requirements and offers advanced

scalability as compared to the brute-pressure approach.

Unusual inputs

I deliberately added elements in the smart contract process testbed that would reason mistakes inside the code of the algorithms. These factors were designed to test the agreement’s resilience and its potential to address uncommon inputs, along with incorrect flows or inconsistent elements. Both brute-pressure and optimized algorithms have been not able to process the smart contract correctly whilst confronted with those wrong factors. However, this means that both algorithms have been capable of apprehending the errors gift within the process and that the integrated blunders detection mechanisms designed paintings as anticipated as they’re capable of figuring out inconsistencies or invalid structures in the smart process, which facilitates making certain the integrity and correctness of the processing.

Security requirements

I introduced multiple security requirements to each element and tested whether the algorithms may want to deal with those complexities. Both the brute-force and optimized versions finished properly, nonetheless, the previous skills increased time and memory utilization because of producing all viable combinations as the computational sources required to method the improved number of security necessities became greater stressful. On the opposite hand, the latter efficaciously controlled the increased time and memory requirements. Using optimized mechanisms, it efficiently recognized and processed only the combinations that no longer create conflicts, thereby decreasing the computational burden.

Table 7.1 presents the measurements of the execution time and the reminiscence utilization of the algorithm. Note that those metrics represent the common values acquired from accomplishing more than one run of every take a look at the case.

Test Case	Execution Time (ms)	Memory Usage (% of 2024 MB)
Realistic Case	44 ms (Opt) / 138 ms (Brute)	24.06 MB(Opt) / 38.07 MB (Brute)
Boundary Conditions	82 ms (Opt) / 194 ms (Brute)	27.16 MB (Opt) / 42.05 MB (Brute)
Unusual Inputs	24 ms (Opt) / 11 5ms (Brute)	10.04 MB (Opt) / 18.07 MB (Brute)
Security Conflict Vulnerabilities	112 ms (Opt) / 302 ms (Brute)	29.46 MB (Opt) / 52.85 MB (Brute)

Table 7.1: Execution Time and Memory Usage Comparison

With a specific focus on real-life scenarios, I employed three realistic cases as a testbed for my research. These cases include the one utilized as a running example in Section 3.5, which involves a smart-contract process for a road misconstruction claim [18]. Addition-

ally, I examined a smart-contract process for hospital teleconsultation [13]. These cases were selected to provide practical illustrations and insights into the application of smart contracts in various domains. In Table 7.2 are reported the results

Test Case	Execution Time (ms)	Memory Usage (MB)
Realistic Case (Ride-Sharing)	32 ms (Opt) / 115 ms (Brute)	15.46 MB (Opt) / 28.06 MB (Brute)
Realistic Case (Road-misconstruction)	50 ms (Opt) / 140 ms (Brute)	20.43 MB (Opt) / 35.46 MB (Brute)
Realistic Case (Hospital teleconsultation)	92 ms (Opt) / 172 ms (Brute)	29.89 MB (Opt) / 46.08 MB (Brute)

Table 7.2: Realistic Case Comparison

In this revised table, each realistic case represents a specific scenario, namely "Ride-Sharing," "Road-misconstruction," and "Hospital teleconsultation." The execution time and memory usage are provided for both optimized (Opt) and brute-force (Brute) approaches. Please note that the execution time and memory usage values are based on the average results obtained during testing. To obtain these results, the code was tested an average of five times, and the following results were obtained. It is important to note that the third case exhibits higher security requirements and a larger number of tasks compared to the others, resulting in increased execution time.

To ensure a rigorous and unbiased evaluation, the final experiment encompasses two distinct cases. Firstly, the activities of a smart contract process (secBPMN2BC) were extended, enabling a comprehensive assessment of the algorithm's performance in terms of memory utilization and execution time. Secondly, a real-world scenario was meticulously crafted, encompassing additional security requirements for each object. Notably, these augmented security measures were designed to prevent any conflicts among the objects.

By meticulously executing these planned experiments, a wealth of invaluable insights and findings were obtained. These results significantly contribute to the advancement of knowledge in the field and reinforce the validity of the research conducted

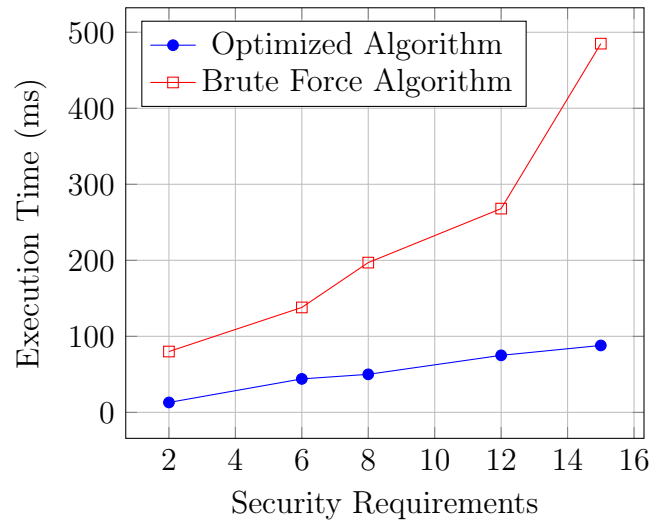


Figure 7.2: Comparison of Execution Time with Increasing Security Requirement

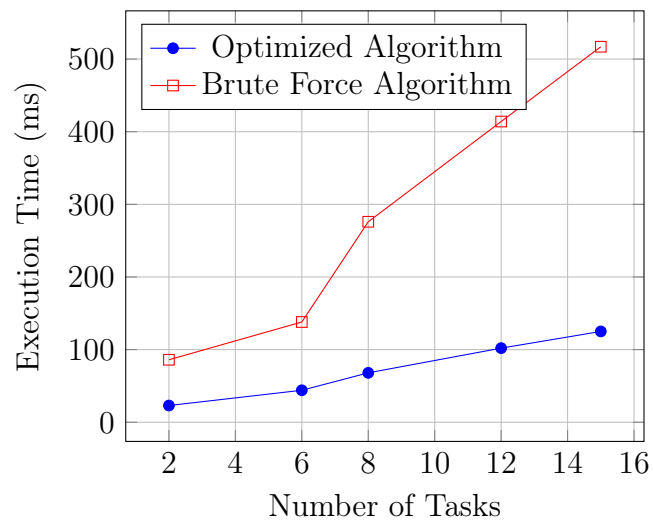


Figure 7.1: Comparison of Execution Time with Increasing Number of Tasks

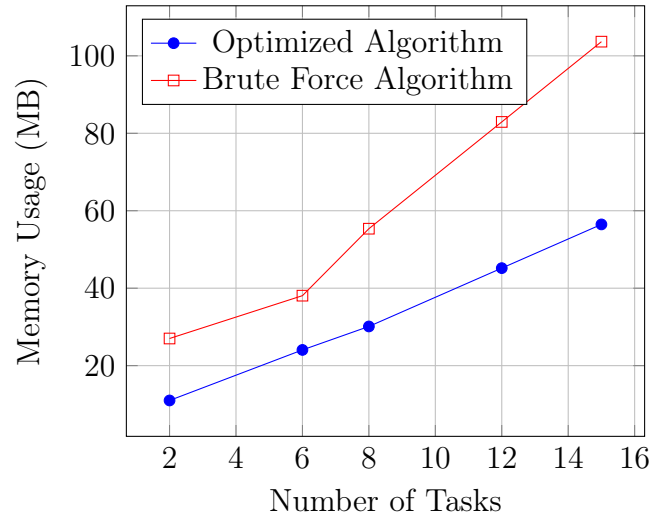


Figure 7.3: Comparison of Memory Usage with Increasing Number of Tasks

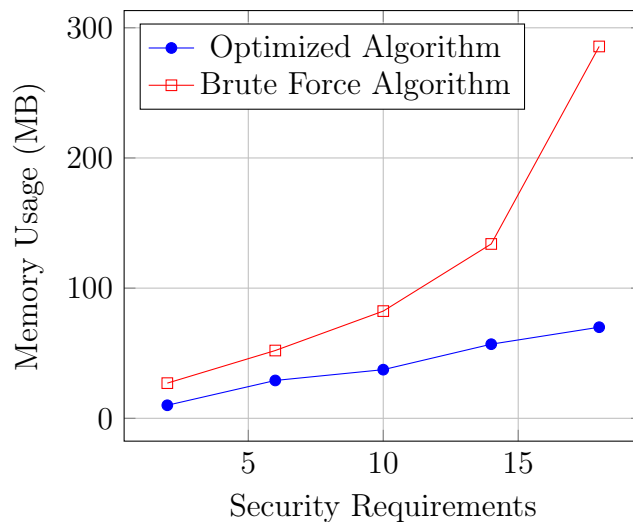


Figure 7.4: Comparison of Memory Usage with Increasing Security Requirement

The provided results, depicted in Figures 7.1, 7.2, 7.3, and 7.4, offer significant insights into the performance of the algorithms under different conditions. Several key considerations can be drawn from the data:

- **Execution Time with Increasing Number of Tasks**

The optimized algorithm consistently outperforms the brute force algorithm in terms of execution time as the number of tasks increases (Figures 7.1). This advantage stems from the optimized algorithm's ability to handle the growing workload more efficiently. As the number of tasks increases, the optimized algorithm exhibits a gradual rise in execution time, indicating a scalable and well-optimized approach. On the

other hand, the brute force algorithm experiences a steeper increase in execution time, highlighting its exponential growth in complexity as it exhaustively explores all possible combinations. The significant time disparity between the two algorithms becomes more evident with larger numbers of tasks, emphasizing the superiority of the optimized algorithm's efficiency and computational effectiveness.

- **Execution Time with Increasing Security Requirement**

Similar to the findings related to the number of tasks, the optimized algorithm consistently outperforms the brute force algorithm as the security requirement becomes more stringent Figure 7.2. As the security requirement increases, both algorithms face additional computations and evaluations. However, the optimized algorithm showcases superior efficiency by managing the increased workload more effectively. It demonstrates a steady and controlled increase in execution time, indicating its ability to handle heightened security requirements without significant performance degradation. In contrast, the brute force algorithm experiences a more pronounced rise in execution time, reflecting the exponential growth in computational complexity as it exhaustively examines all possible combinations. The widening gap between the execution times of the two algorithms with increasing security requirements underscores the optimized algorithm's advantage in efficiently handling demanding security scenarios.

- **Memory Usage with Increasing Number of Tasks**

Examining the memory usage trend in Figure 7.3, the optimized algorithm proves to be more resource-efficient compared to the brute force algorithm as the number of tasks increases. The optimized algorithm's superior memory management is evident from its lower memory consumption, even as the workload grows. It demonstrates an ability to effectively handle and process the increasing number of tasks while optimizing memory usage. Conversely, the brute force algorithm exhibits higher memory usage due to its exhaustive approach, which requires storing and processing a significantly larger number of combinations. As the number of tasks expands, the brute force algorithm's memory usage rises substantially, indicating its reliance on substantial computational resources.

- **Memory Usage with Increasing Security Requirement**

The optimized algorithm continues to showcase its efficiency in memory utilization compared to the brute force algorithm as the security requirement becomes more stringent in Figure 7.4. The optimized algorithm's effective memory management

is evident from its lower memory consumption, even with heightened security requirements. It efficiently handles the additional data and computations required for stringent security scenarios while maintaining a controlled memory footprint. In contrast, the brute force algorithm exhibits a substantial increase in memory usage as the security requirement becomes more demanding. Its exhaustive search approach necessitates storing and processing an exponentially growing number of combinations, resulting in significantly higher memory consumption.

7.4. Discussion of Findings

Based on the result in Table 7.1, we can make a clear distinction between the optimized and brute-force versions of the algorithm in terms of execution time, memory usage, accuracy, and scalability.

As a long way as execution time is concerned, the optimized model consistently outperformed the brute force version in all take a look at instances, showing appreciably faster processing times. This improvement is particularly glaring in the realistic case, boundary situations, and security conflict vulnerabilities, where the optimized version showed a significant reduction in execution time relative to the brute force version. Inside the instances of uncommon inputs, the execution time gap between the two versions has been especially shorter when you consider that both algorithms, as predicted, run into conflict in comparing all of the feasible combinations. Thus, the effects show that the optimized version outperforms the brute-force model. Regarding memory usage, the optimized set of rules constantly confirmed lower memory utilization as compared to the brute force model in all tests. This demonstrates that this version utilizes gadget resources greater correctly, as seen by means of the reduced reminiscence wishes. The distinction in reminiscence utilization is greater pronounced in boundary conditions and protection security vulnerabilities, where brute pressure versions show better reminiscence usage. Consequently, in terms of scaling, optimized variations display extra scalability potential because of reduced reminiscence utilization and faster execution. The updated model has decreased reminiscence requirements and faster processing times, which makes it simpler to control large eventualities. The more reminiscence intake and execution time are used, the optimized and the brute force algorithm could have trouble successfully scaling, especially in these particularly useful resource-intensive eventualities. Validation and evaluation analyses finish that advanced versions of the approach supply full-size enhancements in computing performance and memory utilization at the same time as keeping solution quality. In conclusion, the optimized version of the approach now not only outperforms the brute

pressure variant in phrases of execution time and reminiscence usage but additionally contains a wide array of additional advantages. Its robustness, versatility, security features, ease of renovation, and potential for destiny advancements make it a clear preference for sensible utility in various domain names. The validation of this research endeavor substantiates the achievement through a meticulous analysis of the algorithm's conduct under numerous test conditions. The research specializes in two essential components: the effect of increasing the number of tasks and the incensement of safety requirements.

These precise concerns enhance the clear advantage of the optimized algorithm over the brute force one. The former continuously demonstrates higher scalability, and performance, allowing it to address expanded tasks and stringent security requirements more correctly. These findings align with the anticipated conduct of the algorithms based totally on their respective procedures and computational complexities, highlighting the real superiority of the optimized algorithm in actual-international scenarios. The optimized version of SecBPMN2BC offers distinct benefits for enforcing the global enforcement calculation. With quicker execution time, decrease memory utilization and stronger scalability, it outperforms the brute force algorithm. This performance interprets into a faster computation of the worldwide enforcement parameter, empowering stakeholders with well-timed records for choice-making and warfare avoidance. By harnessing the optimized version, SecBPMN2BC allows more efficient security analysis making sure of strong and effective enforcement of blockchain security requirements.

8 | Conclusions and future developments

This thesis addresses the important goal of finding out which elements of a process can be stored on-chain or off-chain using blockchain technology. By conducting a thorough assessment of the security requirements of the process, evaluating the potential impact on the blockchain structure, and using the implemented algorithms, the study presents a novel approach to improving an existing algorithm for designing smart contracts processes suitable for implementation on a blockchain with smart contracts in SecBPMN2BC.

The algorithms introduced are specifically designed to tackle the challenges arising from conflicting and unsupported security properties in smart contracts within a blockchain framework, by providing a systematic approach to identifying and evaluating the security requirements of each data element in a process.

Specifically, a brute-force algorithm and an optimized one were designed. The former model explores an exhaustive assessment of all viable combinations of security features, taking into consideration a comprehensive analysis of capacity safety situations. Although computationally expensive, this algorithm gives intensive expertise of the alternate-offs concerned in security configurations. The latter algorithm instead incorporates various strategies for improving algorithm performance and efficiency, such as reducing redundant computation and using parallel processing capabilities. This algorithm aims to strike a balance between accuracy and computation between objects, for faster and more efficient security analysis.

An important aspect of this thesis is the achievement of a global enforcement solution for validating the security of smart contracts. This breakthrough was made possible by introducing a new variable, **global enforcement** within the combination of each element that is subjected to a security requirement. The two algorithms employed in this study played a crucial role in accomplishing this objective, with the optimized algorithm serving as the primary method and the brute-force algorithm acting as a validation layer.

The optimized algorithm demonstrated its effectiveness by providing a coherent and ef-

efficient solution to address conflicting and unsupported security properties in smart contracts within a blockchain framework. By incorporating various strategies to enhance performance and efficiency, this algorithm strikes a balance between accuracy and computation speed. Its successful implementation validated the security solution achieved as compared to the brute-force algorithm, which provided a comprehensive analysis of security configurations but at a higher computational cost.

The significance of this thesis lies in its contribution to the field of designing and evaluating smart contract processes on the blockchain. By presenting insights into security requirements and considering both computational performance and exhaustive evaluation.

Looking ahead, the potential applications and impact of this research are expanding. The findings presented in this thesis lay the groundwork for the development of intelligent and adaptable algorithms capable of dynamically adjusting security configurations based on value variables and evolving threats.

Furthermore, the knowledge gained from this study opens doors to exploring other areas where blockchain technology can be leveraged to enhance security and considerations in the smart contract process. The algorithms proposed here can serve as a foundation for further investigations into the integration of emerging technologies such as artificial intelligence and machine learning, leading to even more robust and sophisticated security solutions. A possible future research project could be the introduction of cost variables for each combination of security requirements for elements. Incorporating cost considerations allows for an evaluation of the trade-offs between security measures and financial implications, enabling a more comprehensive analysis of the overall system. This addition ensures that the design and implementation of smart contract processes on the blockchain are not only robust and efficient but also economically viable. By combining comprehensive safety analysis, computational efficiency, and the integration of cost variables, we unlock the true potential of blockchain technology in ensuring the integrity and reliability of smart contracts. Through continuous research and innovation, we are paving the way for a more secure and trusted virtual landscape that benefits both organizations and individuals alike.

Bibliography

- [1] S. Ahmadisheykhsarmast and R. Sonmez. A smart contract system for security of payment of construction contracts. *Automation in Construction*, 120:103401, 2020. ISSN 0926-5805. doi: <https://doi.org/10.1016/j.autcon.2020.103401>. URL <https://www.sciencedirect.com/science/article/pii/S092658052030981X>.
- [2] F. H. Al-mutar, O. N. Ucan, and A. A. Ibrahim. Providing scalability and privacy for smart contract in the healthcare system. *Optik*, 271:170077, 2022. ISSN 0030-4026. doi: <https://doi.org/10.1016/j.ijleo.2022.170077>. URL <https://www.sciencedirect.com/science/article/pii/S0030402622013353>.
- [3] S. Asharaf and S. Adarsh. Decentralized computing using blockchain technologies and smart contracts: emerging research and opportunities: emerging research and opportunities. 2017.
- [4] L. Ballotta, M. R. Jovanović, and L. Schenato. Optimal network topology of multi-agent systems subject to computation and communication latency. In *2021 29th Mediterranean Conference on Control and Automation (MED)*, pages 249–254, 2021. doi: 10.1109/MED51440.2021.9480167.
- [5] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, page 91–96, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450345743. doi: 10.1145/2993600.2993611. URL <https://doi.org/10.1145/2993600.2993611>.
- [6] X. Cao, J. Zhang, X. Wu, and B. Liu. A survey on security in consensus and smart contracts. *Peer-to-Peer Networking and Applications*, 15:1–21, 03 2022. doi: 10.1007/s12083-021-01268-2.
- [7] M. P. Caro, M. S. Ali, M. Vecchio, and R. Giaffreda. Blockchain-based traceability in agri-food supply chain management: A practical implementation. In *2018 IoT Ver-*

- tical and Topical Summit on Agriculture-Tuscany (IOT Tuscany)*, pages 1–4. IEEE, 2018.
- [8] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446, 2017. doi: 10.1109/SANER.2017.7884650.
- [9] H. Chu, P. Zhang, H. Dong, Y. Xiao, S. Ji, and W. Li. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Information and Software Technology*, 159:107221, 2023. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2023.107221>. URL <https://www.sciencedirect.com/science/article/pii/S0950584923000757>.
- [10] L. Gudgeon, P. A. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. Sok: Off the chain transactions. *IACR Cryptol. ePrint Arch.*, 2019:360, 2019.
- [11] T. Hepp, M. Sharinghousen, P. Ehret, A. Schoenhals, and B. Gipp. On-chain vs. off-chain storage for supply- and blockchain integration. pages 283–291. URL <https://doi.org/10.1515/itit-2018-0019>.
- [12] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858, 2016. doi: 10.1109/SP.2016.55.
- [13] J. Köpke, G. Meroni, and M. Salnitri. Designing secure business processes for blockchains with secbpmn2bc. *Future Generation Computer Systems*, 141:382–398, 2023. ISSN 0167-739X. doi: <https://doi.org/10.1016/j.future.2022.11.013>. URL <https://www.sciencedirect.com/science/article/pii/S0167739X22003764>.
- [14] C. Li, B. Palanisamy, and R. Xu. Scalable and privacy-preserving design of on/off-chain smart contracts. 02 2019.
- [15] Y. Li, J. Weng, M. Li, W. Wu, J. Weng, J.-N. Liu, and S. Hu. Zerocross: A sidechain-based privacy-preserving cross-chain solution for monero. *Journal of Parallel and Distributed Computing*, 169:301–316, 2022. ISSN 0743-7315. doi: <https://doi.org/10.1016/j.jpdc.2022.07.008>. URL <https://www.sciencedirect.com/science/article/pii/S0743731522001733>.
- [16] V. Manahov. Cryptocurrency liquidity during extreme price movements: is there a problem with virtual money? *Quantitative Finance*, 21(2):341–360, 2021.
- [17] B. K. Mohanta, S. S. Panda, and D. Jena. An overview of smart contract and use

- cases in blockchain technology. In *2018 9th international conference on Computing, communication and networking technologies (ICCCNT)*, pages 1–4. IEEE, 2018.
- [18] A. Name. testo di partenza, Year Published. URL <https://hdl.handle.net/11311/1226537>. Accessed on Month Day, Year.
- [19] A. Name. testo di partenza, Year Published. URL <https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.18+commit.87f61d96.js>. Accessed on Month Day, Year.
- [20] A. Name. testo di partenza, Year Published. URL <https://docs.soliditylang.org/en/v0.8.9/>. Accessed on Month Day, Year.
- [21] A. Name. testo di partenza, Year Published. URL <https://trufflesuite.com/docs/truffle/how-to/debug-test/test-your-contracts/>. Accessed on Month Day, Year.
- [22] A. Name. testo di partenza, Year Published. URL <https://www.mdpi.com/2079-9292/12/9/2046#>. Accessed on Month Day, Year.
- [23] A. Name. blockchain-council, Year Published. URL <https://www.blockchain-council.org/blockchain/crypto-off-chain-vs-on-chain>. Accessed on Month Day, Year.
- [24] A. Name. on-chain-vs-off-chain, Year Published. URL <https://zebpay.com/blog/on-chain-vs-off-chain#:~:text=0n%2Dchain%20transactions%20take%20significantly,the%20need%20to%20validate%20transactions>. Accessed on Month Day, Year.
- [25] A. Name. on-chain-vs-off-chain, Year Published. URL <https://nextrope.com/smart-contract-attacks-the-most-memorable-blockchain-hacks-of-all-time/>. Accessed on Month Day, Year.
- [26] A. Name. on-chain-vs-off-chain, Year Published. URL <https://cointelegraph.com/learn/what-are-parachains-polkadot-and-kusama-parachains>. Accessed on Month Day, Year.
- [27] A. Name. testo di partenza, Year Published. URL <https://hdl.handle.net/11311/1063708>. Accessed on Month Day, Year.
- [28] Q. Ramadan, D. Strüber, M. Salnitri, V. Riediger, and J. Jürjens. Detecting conflicts between data-minimization and security requirements in business process models. In A. Pierantonio and S. Trujillo, editors, *Modelling Foundations and Applications*,

- pages 179–198, Cham, 2018. Springer International Publishing. ISBN 978-3-319-92997-2.
- [29] A. Rodríguez, E. Fernández-Medina, and M. Piattini. A bpmn extension for the modeling of security requirements in business processes. *IEICE transactions on information and systems*, 90(4):745–752, 2007.
- [30] S. Rouhani and R. Deters. Security, performance, and applications of smart contracts: A systematic survey. *IEEE Access*, 7:50759–50779, 2019. doi: 10.1109/ACCESS.2019.2911031.
- [31] M. Salnitri, F. Dalpiaz, and P. Giorgini. Designing secure business processes with secbpmn. *Software and Systems Modeling*, 16, 07 2017. doi: 10.1007/s10270-015-0499-4.
- [32] M. Salnitri, F. Dalpiaz, and P. Giorgini. Designing secure business processes with secbpmn. *Software and Systems Modeling*, 16, 07 2017. doi: 10.1007/s10270-015-0499-4.
- [33] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243780. URL <https://doi.org/10.1145/3243734.3243780>.
- [34] S. Wang, X. Tang, Y. Zhang, and J. Chen. Auditable protocols for fair payment and physical asset delivery based on smart contracts. *Ieee Access*, 7:109439–109453, 2019.
- [35] G. Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White paper*, 21(2327):4662, 2016.
- [36] L. Zhang, W. Chen, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen. Cbgru: A detection method of smart contract vulnerability based on a hybrid model. *Sensors*, 22(9), 2022. ISSN 1424-8220. doi: 10.3390/s22093577. URL <https://www.mdpi.com/1424-8220/22/9/3577>.
- [37] P. Zhang, J. White, D. C. Schmidt, and G. Lenz. Design of blockchain-based apps using familiar software patterns to address interoperability challenges in healthcare. In *PLoP-24th Conference On Pattern Languages Of Programs*, 2017.
- [38] A. Zohar. Bitcoin: under the hood. *Commun. ACM*, 58:104–113, 2015.

List of Figures

3.1	Graphical annotations for security annotations of SecBPMN2BC [18] . . .	13
3.2	Graphical annotations for Pivity of SecBPMN2BC-ML [18]	15
3.3	Graphical annotations for Enforceability of SecBPMN2BC-ML [18]	15
3.4	Blockchain enforcement rules for SecBPMN2BC security annotations [18] .	18
3.5	Example of SecBPMN2BC diagram for Ride Sharing System	20
5.1	Graphical representation of the Propagate-up phases	35
5.2	Graphical representation of the Propagate-down phases	46
7.2	Comparison of Execution Time with Increasing Security Requirement . . .	72
7.1	Comparison of Execution Time with Increasing Number of Tasks	72
7.3	Comparison of Memory Usage with Increasing Number of Tasks	73
7.4	Comparison of Memory Usage with Increasing Security Requirement	73

List of Tables

4.1	$S_{ConfirmRide}(Audiability)$	27
4.2	$S_{ConfirmRide}(Integrity)$	29
4.3	$S_{ConfirmRide}(Audiability)$	30
4.4	$S_{ConfirmRide}(Integrity)$	30
4.5	$S_{ConfirmRide}(node)$	31
5.1	$S_{node}(ConfirmRide)$	34
5.2	$S_{temp}(ConfirmRide)$	36
5.3	$Subset_{up}(ConfirmRide)$	37
5.4	$Subset_{up}(ConfirmRide)$	37
5.5	$Subset_{up}(ConfirmRide)$	37
5.6	$Subset_{up}(ConfirmRide)$	37
5.7	$S_{node}(Driver)$	38
5.8	$S_{final}(Driver)$	38
5.9	$S_{final}(root)$	45
5.10	$S_{best}(root)_1$	47
5.11	$S_{node}(Driver)_1$	48
5.12	$S_{best}(root)_1$	48
5.13	first combination of $S_{node}(Driver)$ 5.11	48
5.14	$S_{Best}(Driver)_1$	49
5.15	$S_{best}(ConfirmRide)_1$	49
5.16	$S_{best}(ConfirmRide)$	50
6.1	$S_{node}(ConfrimRide)$	54
6.2	$S_{temp}(ConfrimRide)$	55
6.3	$S_{node}(Driver)$	56
6.4	$S_{final}(Driver)$	56
6.5	$S_{final}(root)$	62
6.6	$S_{Best}(root)$	63
6.7	$S_{best}(Driver)$	63

6.8	Best Combination for $S_{best}(Driver)$	64
7.1	Execution Time and Memory Usage Comparison	70
7.2	Realistic Case Comparison	71

Acknowledgements

Ringrazio il mio relatore , Mattia Salnitri, per la sua guida, il suo sostegno e la sua preziosa consulenza durante tutto il percorso della mia tesi. Grazie per il tempo dedicato alla supervisione del mio lavoro, per i consigli e per avermi incoraggiato a superare le sfide che ho affrontato lungo il percorso. La sua fiducia in me è stata un grande stimolo e non posso che ringraziarla per avermi guidato nel completamento di questo importante traguardo accademico.

Desidero ringraziare il mio correlatore, Giovanni Meroni, per aver condiviso le sue conoscenze, esperienze e risorse, che hanno arricchito notevolmente il mio lavoro. Inoltre, volevo ringraziarla per i suoi preziosi commenti, suggerimenti e critiche costruttive, che sono stati fondamentali per lo sviluppo della mia tesi.

