Executive Summary of the Thesis

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

Laurea Magistrale in Computer Engineering - Ingegneria Informatica

Author: Tecla Perenze

Advisor: Prof. Mattia Salnitri

Co-advisor: Prof. Giovanni Meroni

Academic year: 2022-2023

1. Introduction

Smart contracts are blockchain-based algorithms that execute when certain criteria are met, eliminating the need for intermediaries and third parties in transaction processes. Once deployed, smart contracts are transparent and immutable, making their results tamper-proof and unchangeable [5].. The automated nature of smart contracts brings several advantages compared to traditional contracts, including improved efficiency and cost-effectiveness [13]. However, it is crucial to prioritize security due to the potential financial and legal consequences that can arise from security vulnerabilities [3].

To mitigate these risks, blockchain technology provides a secure and transparent environment for the execution of smart contracts. The decentralized and immutable nature of nodes ensures that transactions are verified by each node, updating the blockchain to reflect the modifications made by the smart contract [11]. However, blockchain technology may not always satisfy all security requirements, and there may be conflicts between some security requirements and aspects of blockchain technology. 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.

The primary goal of this thesis is to identify which elements of a given process can be stored on-chain, rather than off-chain through an analysis of security requirements, indeed, the study assesses the process's security needs and identifies the best components that can be stored using blockchain technology. The thesis presents a novel approach to improve an existing algorithm [7] using SecBPMN2BC, a modeling language that guides process modelers in designing secure business processes suitable for implementation on a blockchain using smart contracts. Two algorithms have been developed within this framework: a brute force algorithm and an optimized algorithm. Both are aimed at determining whether an element should be placed on the blockchain or kept off-chain through an analysis of the security requirements associated with the smart contract element. The focus is on finding a comprehensive solution through two different approaches that transcend the limitations of the original algorithm, which only offered local solutions.

2. State of Art

Since the proliferation of both blockchain and smart contracts has recently increased, the demand for secure data procession and storage is exponentially grown, thus the problem of deciding whether to put an element of the smart contract on-chain or off-chain is nowadays a critical decision [15]. Several approaches have been implemented to address this theme: machine learning, dynamic and static analysis, and Cross-chain solutions.
More in detail, Machine learning-based techniques can involve training methods on past smart contract transactions to predict the ideal storage location for the new ones, so as to give a great enhancement to the decision-marketing process for on-chain or off-chain storage [9]. Nevertheless, it does not represent a fail-safe solution. Machine learning algorithms can be subject to inaccuracies and the amount of training data needed can be highly taxing [8]. Additionally, these techniques do not address the security problems to identify all the possible conflicts and vulnerabilities, as the model itself is at risk, so these performing techniques should be used in conjunction with other measures to guarantee security assessment as well as security and integrity for smart contracts.
Other topics in the literature that are promising to optimize the smart contract code for deciding where is better to store elements of the smart contract are the so-called dynamic and static analysis framework and algorithms [9]. Nevertheless, due to the ongoing evolution of smart contracts and blockchain architecture, these strategies might not always offer the best answer. It is, however, not deniable that the subject of smart contract security and storage optimization has advanced from the results of these discoveries.
The framework "The Gasper" [2], for instance, is used to evaluate the security of smart contracts. It offers a thorough testing strategy that addresses a variety of security flaws, is open-source, adaptable, and necessitates substantial processing power. Another useful security tool called "Securefy" [12] may greatly increase the security of smart contracts. The black mark is that it may be expensive, and might only work with a restricted number of smart contract platforms or languages [1].
Cross-chain solutions may help researchers to facilitate the transfer of assets and data across different blockchain networks [4], nevertheless it is not sufficient to solve the security requirement problems but numerous projects and initiatives have been actively working to develop solutions in this domain, an example is Polkadot [14], it is a multi-chain platform designed to enable cross-chain communication as allow different blockchain network to connect and share information through a unified protocol. It can thus facilitate the decision of whether an element should be placed on-chain or off-chain through its Paracahain structure. To be clearer a Parachain is a specialized chain connected to the Polkadot Relay Chain, which acts as the main network for coordination and consensus [6]. Through this platform, developers may construct and opt to deploy their smart contracts on particular Parachains, giving them the freedom to choose whether certain smart contract components should be processed on-chain or off-chain. It is worth nothing to say that the entire decision should not rely on human capability as an automated process will significantly reduce the errors, especially when it comes to security, a tool that analyzes all the security requirement is thus necessary to guide the better organization of element of smart contract to make it temper-proof.

3. Baseline

This chapter will provide a detailed overview of the algorithms employed to determine the appropriate execution location (on-chain or off-chain) for specific components within a smart contract. Through an analysis of all security requirements assigned to the element of the smart contract taken into examination, it manages to find a globally optimal solution that helps the developer with the decision of putting an element on a chain or off-chain. To achieve this result SecBPMN2BC has been implemented to model the smart contract process that is given in input to the overmentioned algorithms. SecBPMN2BC is an extension of the widely used Business Process Model and Notation (BPMN) [10] architecture designed to address security concerns in modeling and implementing blockchain technology business processes. 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. To help the reader to understand the detail of the algorithm and its workflow, a running example in Figure 1 has been implemented in order to guide through the whole thesis and is tailored specifically for a ride-sharing platform. It shows the smart-contract process that starts from the client's request and terminates with the completion of the ride, without excluding all events that may occur. The smart contract has been implemented using SecBPMN2BC, in fact as it is possible to notice by the structure in Figure 1, three pools identify the main "participant" of the process, including the Ride-Sharing Platform, the Driver, and Payment. In each pool, tasks that represent the activities are presented, and each of them is connected throw sequence flows if two Tasks are in the same pool, with sequence messages if are in two different pools.

What is important to notice is that both Tasks and Data Objects are subjective to security requirement that belongs to the business process and model notation language. Figure 2 presented the security requirement in secBPMN2BC.

To decide the storage of the element of the smart contract on the blockchain it is crucial to take into account the kind of blockchain being employed. While private blockchains are controlled by a central organization or a small group of people, public blockchains are decentralized and enable everyone to access information. OnChainModel, OnChainExecution, OnChainData, and BlockchainType are all properties that determine the execution logic of an activity on-chain or off-chain. OnChainExecution specifies whether an activity will be executed on-chain via a smart contract or off-chain. OnChainData specifies how data associated with a message or data object state will be stored and validated. BlockchainType specifies whether the on-chain portion of the process will be executed on a public or private blockchain.

Rules for getting the ideal fusion of security elements can be derived. In particular, the corresponding process elements are identified during the analysis of each rule. There are several sets of property values identified for each element that satisfies the rule's requirements. The characteristics of the process element itself and those of its relatives or related components are included in these sets. Ultimately, a label is assigned to each combination of characteristics indicating how strictly the rule is enforced by the blockchain's security system. The designations are *native* when the blockchain fully or partially enforces the property, *possible* when it does so partially, and *no enforcement* when it offers no assistance. In figure 3 are presented all the rules that can be derived.

4. Algorithmic Framework

The architecture and security features of blockchain require careful consideration of which security properties can be supported on-chain and which must be executed off-chain, as failure to address these conflicts and limitations can expose smart contracts to significant security vulnerabilities. To tackle this problem, the algorithms presented aim to identify and optimize the set of security properties that can be executed on-chain, prioritizing the security requirements that contribute to the highest possible level of security for the smart contract. By doing so, the algorithms mitigate the risks associated with conflicting properties and enhance as they identify a combination of security properties of SecBPMN2BC for each node in a smart contract, taking into account a global variable called **Global Enforcement**. This numerical variable helps to determine if an element of a smart contract should be executed on-chain or off-chain. In particular, the brute force algorithm explores all possible combinations of security properties, while the optimized algorithm reduces the number of combinations to explore through a heuristic approach.

4.1. Global Enforcement

This parameter is a numerical value given to each set of security rule combinations that aims to indicate the global level of security that is enforced by the blockchain's security properties. The calculation of Global Enforcement in the context of combination rules takes into account the local enforcement value of the specific combination:

- When a combination possesses native enforcement, which implies an inherent and

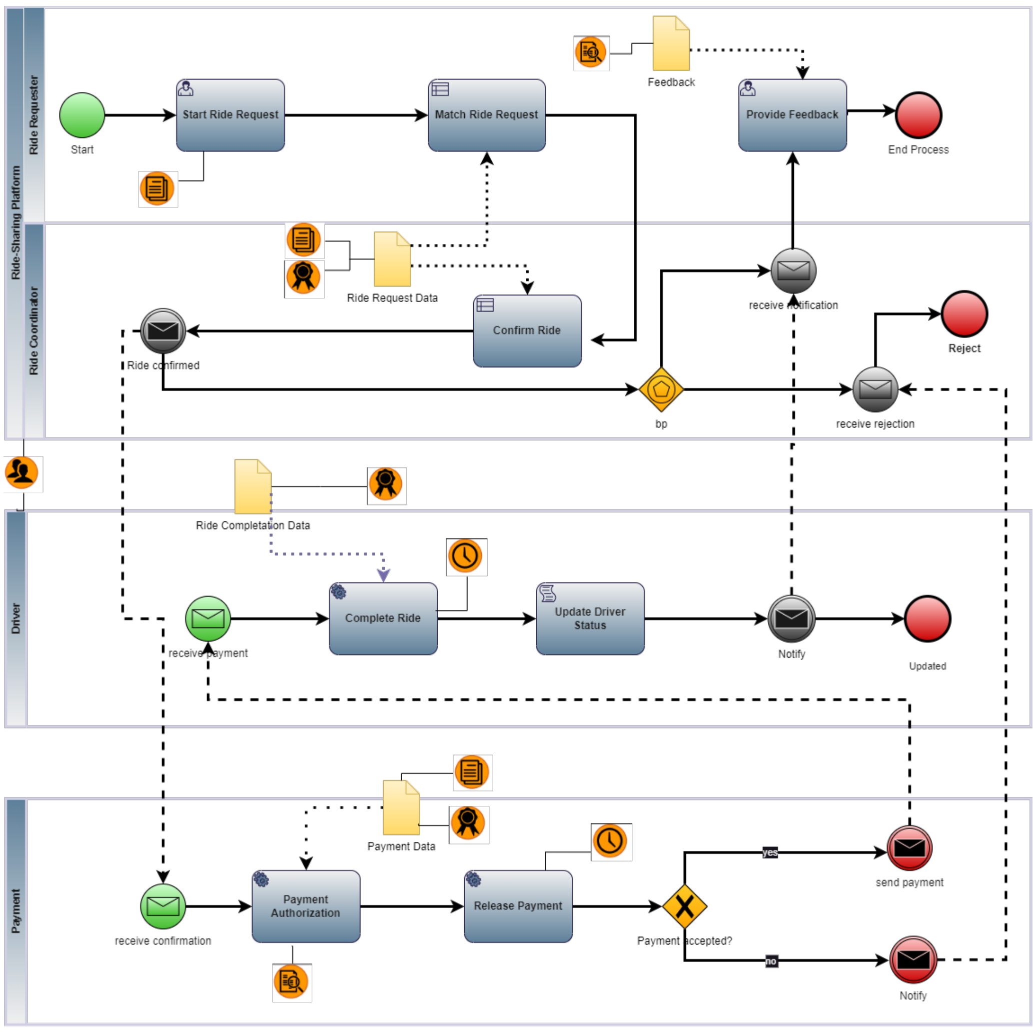


Figure 1: Example of SecBPMN2BC diagram for Ride Sharing System

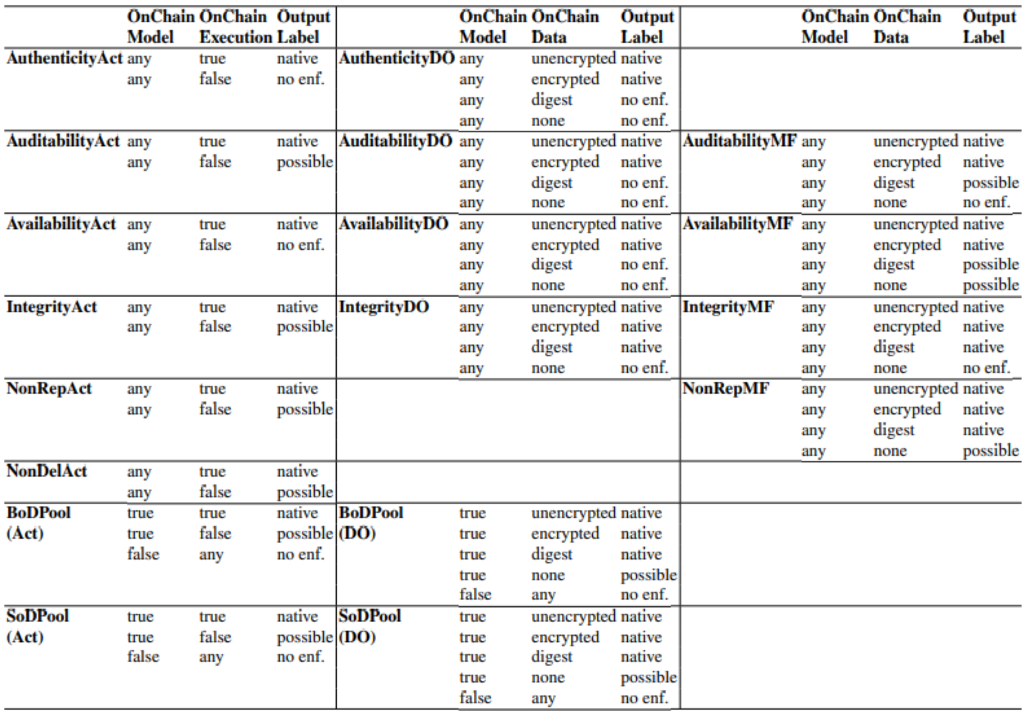
● Auditability	● Separation of duties	● **Privity - public**	● **Enforceability of the control-flow**
● Authenticity	● Bind of duties	● **Privity - static**	● **Enforceability of decisions - public**
● Availability	● Non Delegation	● **Privity - private**	● **Enforceability of decisions - private**
● Integrity	● Privacy	● **Privity - strong dynamic**	● **Enforceability of decisions - user defined**
● Non Repudiation		● **Privity - weak dynamic**	● **On-chain** (To be used on P2 and P3)

Figure 2: Graphical annotation of SecBPMN2BC Security Requirements [7]

automatic enforcement mechanism, the Global Enforcement value is assigned as 1, indicating complete and full enforcement.

- If a combination has no enforcement, the Global Enforcement value is assigned as 0
- In cases where the combination has possible enforcement, the Global Enforcement value is assigned a value between 0 and 1, representing the likelihood or probability of the combination being enforced.

Regarding the "possible" case, it is important to state the range of values that can be assigned to the Global Enforcement in order to allow developers to prioritize and compare different requirements based on their respective enforcement capabilities, make informed decisions, and allocate resources accordingly. Consequently, it is crucially important to analyze all scenarios where different levels of enforcement can be achieved for each security requirement. For



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

Figure 3: Blockchain enforcement rules for SecBPMN2BC security annotations [7]

instance, security requirements such as BOD, SOD, and Non-Delegation all share the characteristic that if their enforcement level is possible, the entire activity will be executed off-chain. As a result, the range of values assignable to Global Enforcement for these requirements will be narrower compared to other security annotations like Auditability, Availability, Integrity, and Non-repudiation. These latter requirements benefit from partial protection by the inherent structure of the blockchain. Consequently, the range of values for Global Enforcement in these cases is higher, approaching the naive case. Ultimately, the user retains the authority to determine the exact value for the Global Enforcement within the designated ranges, based on their specific needs. This empowers them to make customized choices aligned with their requirements and priorities.

4.2. Algoritmic description

The algorithm utilizes a tree structure to depict the process model, where each element within the smart contract is associated with a rule set denoted as $S(node)$. This set represents the individual security requirement properties of the corresponding node. The root element represents the *blockchainType* property, that is the process definitions. Pools are intermediate elements that can have their $onChainModel$ attribute set, as well as intermediary items like subprocess activities. Tasks, Data Objects, and Messages are the process child components. The $onChainData$ attribute can be set for leaf items such as Data Objects and Messages, while $onChainExecution$ is for the Tasks. Each element has its specific property that is composed of its local combination and its indirect ones. More in detail "If a property belongs to a particular process element, we say that the property value assignment directly holds for that element. If a property belongs to an ancestor of an element rather than the element itself, we say that a property value assignment for that element indirectly holds" [7]. For example, the Confirm Ride task node from the Running Example in Section 1 subjected to the Audiability security requirement acquires a set of property combinations, each comprising three security requirements: $onChainExecution$ for the local node, $onChainModel$ for its parent, and $blockchainType$ for its ancestors. 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. To visualize the structure described above, Figure 4 provides a simplified graphical representation of the process model.

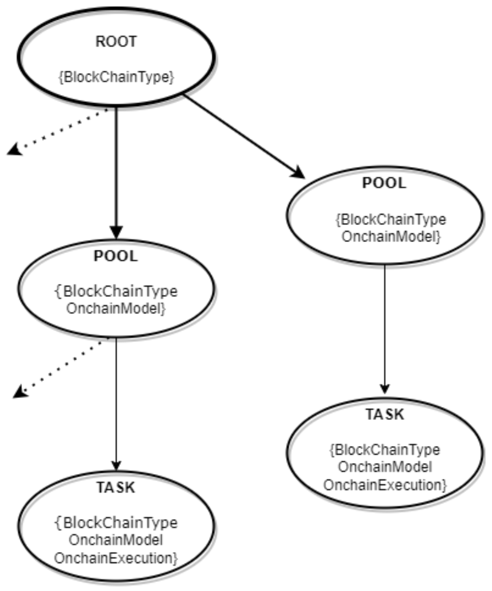

ROOT
{BlockChainType}
POOL
{BlockChainType OnchainModel}
POOL
{BlockChainType OnchainModel}
TASK
{BlockChainType OnchainModel OnchainExecution}
TASK
{BlockChainType OnchainModel OnchainExecution}

Figure 4: Graphical process representation

4.3. Deriving Local Combination

A single node within the system can be subject to multiple security requirements, each generating its own set of constraints. To establish a coherent and comprehensive set $S(node)$ of combinations for a node, it is necessary to merge the individual security requirement sets $S_{Rn}(node)$. However, this merging process can introduce conflicts among the requirements when different security rules impose contradictory or incompatible conditions. Conflicts arise when two combinations, $C_1 = \langle [P_k]1, le1, gle_1 \rangle$ from the rule set $S_{R1}(node)$ and $C_2 = \langle [P_k]_2, le_2, gle_2 \rangle$ from the rule set $S_{R2}(node)$, associated with different security requirements for the same node, exhibit the same property pair $[P_k]_1$ and $[P_k]_2$, but have different local enforcement values (le) or global enforcement values (gle) ($le_1 \neq le_2$ or $gle_1 \neq gle_2$). These conflicts arise due to divergent enforcement values for the same property pair within different sets of rules representing distinct security requirements associated with the same node.
To address this problem, the approach used focuses on prioritizing rules that impose fewer limitations on specific security needs. This is achieved by evaluating the local enforcement values le for each combination and selecting the one with the lowest value.
By selecting the combination with the lowest enforcement value, the strategy aims to minimize the limitations and constraints placed on the security need. This approach allows for a more flexible and accommodating solution, prioritizing combinations that provide fewer restrictions while still meeting the necessary security standards. To find a compromise between conflicting enforcement values and gain a global perspective, the average of the matching values from the two sets is computed to update the global enforcement value (gle).

$$gle = \frac{gle_1 + gle_2}{2} \tag{1}$$

This approach produces a global enforcement value that reflects a compromise between the conflicting goals, ensuring that no one set of rules predominates over the other.
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.

4.4. Structure of the algorithm

The algorithms consist of two phases: the bottom-up and top-down phases. In the bottom-up phase, the algorithm propagates the set of combinations from the child nodes to their parent node. The objective is to update the parent node's set of combinations by incorporating only those that do not result in conflicts, following the method explained in Section 4.3. This ensures that the parent node maintains a consistent and conflict-free set of combinations based on the combinations received from its child nodes.
In the top-down phase, a single combination is chosen from the obtained set. This selected combination represents the highest level of security enforcement and is then transmitted downward in the second phase to eliminate any incompatible combinations determined in the first phase.
There are two implemented versions of the algorithm that share the same structure but differ in the criteria used to identify all potential combinations and select the best one

5. Bruteforce strategy

The brute force approach is a technique used to solve optimization problems by exhaustively evaluating all possible combinations of elements. It consists of two stages, bottom-up and top-down, where every potential combination is considered. This algorithm systematically explores each solution without using optimization techniques to narrow down the search space.

5.1. Propagate up Bruteforce

The propagateUp algorithm (shown in pseudocode in Algorithm 1) is designed to propagate combinations from leaf nodes to root nodes in a hierarchical structure using a brute-force strategy. Its main objective is to determine the ultimate set of admissible combinations by iteratively propagating and constraining them as necessary.
When propagating constraints from child nodes to parent nodes, it is crucial to remove the local properties specific to each current node. This ensures that conflicts are avoided and the parent node's constraints are appropriately enforced. To achieve this, the algorithm creates a temporary set called S_{temp}, which contains all combinations of the current node excluding its local properties.
However, during the removal of local properties, duplicate combinations can arise within S_{temp}, leading to conflicts. To address this issue and maintain constraint integrity, the algorithm generates subsets within a set called S_{up}. Each subset represents a unique combination and is derived by considering all possible cases and combinations. By splitting S_{temp} into subsets without repetition using the *splitCombination* function, the algorithm guarantees that each combination appears only once within S_{up}, effectively eliminating conflicts caused by duplicate combinations.
To successfully propagate constraints from child nodes to parent nodes, the subsets of S_{up} need to be merged with the parent node's constraint set. This merging process ensures that the child constraints are incorporated into the parent while considering all possible combinations.
The algorithm accomplishes this by invoking the *constrainSet* function. This function compares each combination from the subsets with the parent constraint set and adds the combination to the final set, denoted as S_{final}, only if it does not conflict 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}. This process of constraint propagation and merging continues recursively throughout the hierarchical structure until it reaches the root node. At each level, the algorithm progressively refines and narrows down the set of admissible combinations. Finally, the algorithm produces the final set of admissible combinations for the entire hierarchical structure, represented by S_{root} and its corresponding relative S_{final}.

Algorithm 1 propagateUp

```
Input: SecBPMN2BC node: element
Output: $S_{\text{final}}$ : all possible admissible property
        value combinations for parent element
$S_{\text{temp}}$ = newSet()
for $C_j$ in $S_{node}$ do
    $C_{temp}$ = newCombination()
    for $P_k$ in $C_j$.properties do
        if $P_k$.name not in node.properties then
            $C_{temp}$.properties.add($P_k$)
    $S_{\text{temp}}$.add($C_{temp}$)
if not node.isLeaf then
    for child in node.children do
        $S_{parent}$ = newSet();
        $S_{up}$ = newSet();
        $S_{final}$ = newSet();
        for $C_{child}$ in propagateUp(child) do
            $C_{parent}$ = newCombination()
            for $P_{child}$ in $C_{child}$.properties do
                if $P_{child}$ not in node.properties then
                    $C_{parent}$.properties.add($P_{child}$)
            $S_{parent}$.add($C_{parent}$)
        if $S_{parent}$ is not Empty then
            $S_{up}$.addAll(splitCombination($S_{parent}$))
        if $S_{up}$ is not Empty() then
            $S_{final}$ = constrainSet($S_{up}$, $S_{\text{temp}}$)
return $S_{final}$
```

5.2. Propagate Down Bruteforce

In this phase, the algorithm aims to derive the best combination from each set in terms of security. Its objective is to select a combination that maximizes security enforcement and propagate it from the root to the leaf nodes, constraining each set accordingly.
Using a brute-force approach, the algorithm considers one combination at a time for each set and propagates it down the tree. After the propagate-up phase, the resulting root set

S_{root} is influenced by the constraints imposed by its children. The algorithm iteratively selects each combination of the root set and propagates it down, constraining each set through the *constrain* function. It removes any incompatible combinations from the child set with the selected combination, resulting in the set S_{best}.
When a node is not a leaf, the algorithm chooses one combination at a time the from the previously constrained set as the current best combination. This combination is saved and propagated to the child nodes. When a node is a leaf, a different method is employed to determine the optimal combination (see Algorithm 3). The set S_{best} obtained by constraining the set with the best combination passed down by the parent node is filtered in order to select the combination that that maximizes security enforcement. It first selects only the combinations with the maximum local enforcement (filtered with *native* value), and if none exist, it filters based on *possible* values. If still no combinations are found, it includes combinations with *no_ enf* values. Finally, among the filtered combinations, the one with the highest Global Enforcement is selected, and the S_{best} is updated with this best combination for the current iteration.
Through this iterative process, the algorithm systematically evaluates all possible combinations within each set, selects the optimal combination at each step, and propagates it down to the leaf nodes.

Algorithm 2 propagateDown

```
input  : SecBPMN2BC node: node
if node.parent then
    $S_{node}$=constrain($S_{best}$,S[node.parent])
    for $C_j$ in $S_{node}$ do
        $C_{best} = C_j$
        $S_{best}$ = newSet($C_{best}$)
    for child in node.children do
        propagateDown(child)
if node is leaf then
    $S_{node}$=constrain($S_{node}$,S[node.parent])
    $C_{best}$ = getBestCombination(node)
    $S_{node}$ = newSet($C_{best}$)
```

Algorithm 3 getBestCombination

```
input  : SecBPMN2BC node
output: Combination $C_{best}$: combination providing
        maximum enforcement
$S_{final}$=getMaxGle(node,'native')
if $S_{final}$.size=0 then
    $S_{final}$=getMaxGle(node,'possible')
if $S_{final}$.size=0 then
    $S_{final}$=getMaxGle(node,'no_enf.')
else
    $S_{final}$.size=0
err.raise(node,'Conflict detected')
```

6. Optimized Strategy

The optimized approach is a more efficient algorithm that aims to find the best combination of elements. It selectively focus on the most promising combinations, eliminating unnecessary computations and reducing search space. It consists of two distinct phases: a bottom-up phase and a top-down phase. Each phase provides a detailed description of each phase.

6.1. Propagate up optimized

The optimized propagateUp algorithm (Algorithm 4) operates within a hierarchical structure, where nodes represent elements with associated properties. Its main goal is to propagate property combinations from child nodes to their parent node, ensuring that the parent node only includes properties not already present in its children.
At a high level, the algorithm begins similarly to the brute force version by initializing an empty set, S_{temp}, to temporarily store combinations. However, the key difference lies in handling conflicts. In this optimized version, the *Constraint* function is invoked on S_{temp} to eliminate conflicts by selecting combinations with the lowest local enforcement value and updating the global enforcement accordingly.
The resulting set is then propagated up to the parent node, and the parent node undergoes a similar constraint process to avoid conflicts.
Finally, the algorithm returns S_{final}, which represents all possible valid combinations of property values for the parent node within the hierarchical structure

Algorithm 4 propagateUp

```
Input: node: GMTNode
Output: $S_{final}$ = all possible admissible property value combinations for parent node
$S_{temp}$ = newSet()
for $C_j$ in S(node) do
    $C_{temp}$ = newCombination()
    for $P_k$ in $C_j$.properties do
        if $P_k$.name not in node.properties then
            $C_{temp}$.properties.add($P_k$)
    $S_{temp}$.add($C_{temp}$)
if not node.isLeaf then
    $S_{parent}$ = newSet()
    for child in node.children do
        $S_{up}$ = propagateUp(child)
        for $C_{child}$ in $S_{up}$ do
            $C_{parent}$ = newCombination()
            for $P_{child}$ in $C_{child}$.properties do
                if $P_{child}$ not in node.properties then
                    $C_{parent}$.properties.add($P_{child}$)
            $S_{parent}$.add($C_{parent}$)
        if $S_{up}$ is not empty then
            $S_{final}$ = constrain($S_{up}$, $S_{temp}$)
return $S_{final}$
```

6.2. Propagate down optimized

In the optimized Propagate Down phase (Algorithm 5), a more efficient approach is taken compared to considering all combinations. Instead, the algorithm focuses on selecting the best combination for each node, maximizing security enforcement.

The algorithm starts at the root node and iteratively chooses the best combination from the available options. This selected combination is then propagated down to the child nodes. To ensure compatibility, a *Constraint* function is used to filter out any incompatible combinations, resulting in a constrained set for the next iteration.

Similar to Algorithm 3 used for leaf nodes in the brute force algorithm, the obtained set from the parent node is filtered based on the same criteria. This allows the algorithm to choose the best combination that maximizes enforcement for each set, which is then propagated down the tree.

By following this iterative process, the algorithm systematically evaluates and selects the optimal combination for each node, efficiently propagating it down to the leaf nodes. This optimized approach reduces computational complexity and ensures a focus on maximizing security enforcement throughout the entire tree.

Algorithm 5 Propagate Down

```
Input: node - current element
Output: Propagate constraints down the tree
if node.parent then
    $S(node)$ = constraint($S_{\text{node}}$, $S_{\text{temp}}$)
    for $C_j$ in $S(node)$ do
        $C_{\text{best}}$ = getBestCombination(node)
        $S_{\text{best}}$ = newSet($C_{\text{best}}$)
        for each child in node.children do
            propagateDown(child);
```

7. Validation

A comparison was conducted between the brute force version and the optimized version, focusing on their characteristics, performance, and evaluation results. The experimental setup involved using the SecBPMN2BC modeling language within the Eclipse development environment to test both algorithms on the same set of smart contracts. The set included realistic cases, boundary conditions, unusual inputs, and security conflict vulnerabilities. The results are presented in Table 1.

The evaluation criteria encompassed execution time, memory usage, accuracy, robustness, and scalability. The analysis of the results revealed that the optimized version outperformed the brute force approach in various aspects. Particularly in boundary situations and security conflict vulnerabilities, the optimized SecBPMN2BC model consistently exhibited faster execution times and lower memory usage.

The results of experiments on the optimized and brute force algorithms are presented in Figures 6, 5, 7, and 8. These graphs illustrate the effects of increasing numbers of tasks and security requirements on execution time and memory usage.

Specifically, the optimized algorithm demonstrates a more gradual increase in execution time as the number of tasks grows compared to the brute force algorithm. It also maintains a steady execution time across different security requirements, whereas the brute force algorithm experiences a more pronounced rise due to its exponential complexity. Additionally, the optimized algorithm consistently requires less memory than

Test Case	Execution Time (ms)	Memory Usage (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 1: Execution Time and Memory Usage Comparison

the brute force algorithm for the same number of tasks and exhibits lower memory usage even under heightened security requirements.
The findings clearly indicate that the optimized algorithm surpasses in terms of performance and control. It demonstrates superior effectiveness in managing expanded tasks and security requirements. Notably, when it comes to computing Global Enforcement parameters, the optimized version outperforms the brute force algorithm, resulting in improved decision-making capabilities. Moreover, the optimized algorithm enhances security analysis, ensuring the reliable enforcement of blockchain security requirements.

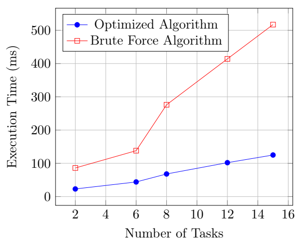


Figure 5: Comparison of Execution Time with Increasing Number of Tasks

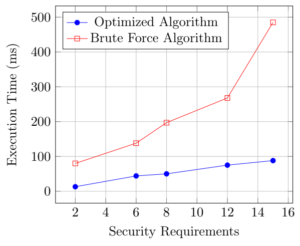


Figure 6: Comparison of Execution Time with Increasing Security Requirement

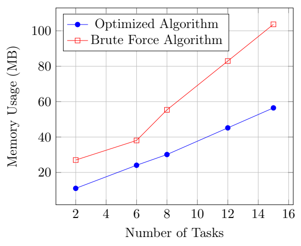


Figure 7: Comparison of Memory Usage with Increasing Number of Tasks

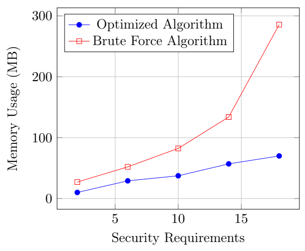


Figure 8: Comparison of Memory Usage with Increasing Security Requirement

8. Conclusion and future developments

This thesis aims to determine which elements of a process can be stored on-chain or off-chain using blockchain technology. It presents a novel approach to improving existing algorithms for designing smart contract processes suitable for implementation on a blockchain with smart contracts in SecBPMN2BC. The algorithms introduced are designed to tackle challenges arising from conflicting and unsupported security properties in smart contracts within a blockchain framework. Two algorithms were designed: a brute-force algorithm and an optimized one. The brute-force algorithm explores an exhaustive assessment of all viable combinations of security features. The optimized algorithm aims to strike a balance between accuracy and computation between objects, resulting in faster and more efficient security analysis.

The achievement of a Global Enforcement solution for validating the security of smart contracts was made possible by introducing a new variable, Global Enforcement, within the combination of each element subjected to a security requirement. The optimized algorithm demonstrated its effectiveness by providing a coherent and 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.
The significance of this thesis lies in its contribution to the field of designing and evaluating smart contract processes on the blockchain. The findings laid the groundwork for the development of intelligent and adaptable algorithms capable of dynamically adjusting security configurations based on value variables and evolving threats. 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.
Future research projects could include introducing cost variables for each combination of security requirements for elements, allowing for an evaluation of the trade-offs between security measures and financial implications. 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, this research unlocks the true potential of blockchain technology in ensuring the integrity and reliability of smart contracts.

References

[1] Salar Ahmadisheykhsarmast and Rifat Sonmez. A smart contract system for security of payment of construction contracts. *Automation in Construction*, 120:103401, 2020.

[2] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong 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.

[3] Thomas Hepp, Matthew Sharinghousen, Philip Ehret, Alexander Schoenhals, and Bela Gipp. On-chain vs. off-chain storage for supply- and blockchain integration. pages 283–291.

[4] Yuxian Li, Jian Weng, Ming Li, Wei Wu, Jiasi Weng, Jia-Nan Liu, and Shun Hu. Zerocross: A sidechain-based privacy-preserving cross-chain solution for monero. *Journal of Parallel and Distributed Computing*, 169:301–316, 2022.

[5] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish 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.

[6] Author Name. on-chain-vs-off-chain, Year Published. Accessed on Month Day, Year.

[7] Author Name. testo di partenza, Year Published. Accessed on Month Day, Year.

[8] Author Name. testo di partenza, Year Published. Accessed on Month Day, Year.

[9] Sara Rouhani and Ralph Deters. Security, performance, and applications of smart contracts: A systematic survey. *IEEE Access*, 7:50759–50779, 2019.

[10] Mattia Salnitri, Fabiano Dalpiaz, and Paolo Giorgini. Designing secure business processes with secbpmn. *Software and Systems Modeling*, 16, 07 2017.

[11] Hamed Taherdoost. Smart contracts in blockchain technology: A critical review. *Information*, 14(2), 2023.

[12] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin 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.

[13] Shangping Wang, Xixi Tang, Yaling Zhang, and Juanjuan Chen. Auditable protocols for fair payment and physical asset delivery based on smart contracts. *Ieee Access*, 7:109439–109453, 2019.

[14] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White paper*, 21(2327):4662, 2016.

[15] Lejun Zhang, Weijie Chen, Weizheng Wang, Zilong Jin, Chunhui Zhao, Zhennao Cai, and Huiling Chen. Cbgru: A detection method of smart contract vulnerability based on a hybrid model. *Sensors*, 22(9), 2022.