POLITECNICO DI TORINO

Master's Degree in Computer Engineering
a.a. 2023-2024
Degree Session April 2024

Distributed Verifier

Smart Contract for verifying ECDSA signatures on multiple curves

Supervisors

Prof. D. Bazzanella

Candidate

Cristian Brunetto

Tutor
LINKS FOUNDATION
Ing. D. Margaria

Summary

The blockchain is an innovative technology based on a distributed ledger shared between all nodes involved in the network. The nodes communicate with each other and manage data, which are transactions, according to a specific communication protocol. The blockchain allows transactions to be validated and stored in the distributed database shared across all the decentralized nodes. The transactions are public so that anyone can retrieve and verify them. Additionally, the transactions recorded in the distributed ledger are stored permanently and become immutable. The recent distributed ledger technology enables the deployment of smart contract onto the network nodes. Smart contracts are distributed apps (dApp) executed from all the nodes of the peer-to-peer network, that compute and agree on the results validating them. Every smart contract transaction requires a payment that is charged to the caller with an amount which is proportional to the complexity of the function called. The nodes use digital signatures computed over elliptic curves to verify transactions. Every blockchain employs a specific curve, for instance, Bitcoin and Ethereum use the SECP256K1. In this kind of environment is possible to verify only transactions that have a fixed structure and are signed on a specific curve.

This work explores the feasibility of the creation of a smart contract able to verify the signature of a generic data calculated over multiple curves without being constrained to a specific blockchain's curve. Although there are some challenges in implementing signature verification in this environment, this type of smart contract has been implemented and works correctly. Additionally the smart contract must be tested for compliance with NIST guidelines. Therefore, various tests have been conducted to ensure its accuracy in meeting the standards. Moreover, tests have been conducted to determine the financial cost and the time required for verification. Finally, a possible use case of the smart contract is presented, highlighting the advantages of blockchain over a centralised environment.

Acknowledgements

Questa tesi rappresenta la conclusione del mio percorso universitario, vorrei quindi ringraziare tutti coloro che mi hanno accompagnato in questa esperienza.
In particolare vorrei dire grazie ai miei genitori, che mi hanno permesso di studiare e supportato lungo questo cammino. Un grazie a mia madre che, come solo una mamma sa fare, ha saputo starmi accanto durante questi anni, e a mio padre che mi ha insegnato la resilienza, a non mollare davanti alle avversità, perchè le cose belle si ottengono con il lavoro e l'impegno. Un particolare ringraziamento va anche a mio fratello Manuel, che nonostante i molti anni di differenza ha saputo capire la difficoltà e l'impegno richiesto in questo percorso ed è sempre stato al mio fianco.
Vorrei ringraziare in maniera davvero sentita i miei nonni che, in tutti questi anni, mi sono sempre stati vicini. Non dimenticherò mai i pranzi che nonostante il poco prevviso mi preparavano prima delle lezioni e in particolare ciò che mi diceva mio nonno: 'Forza Cri, mangia in fretta, se no fai tardi'. Per lui il mio percorso universitario era davvero importante. Purtroppo a causa del covid non ci sei più e non hai potuto vedermi ottenere la laurea Triennale e adesso la Magistrale. In ogni caso spero che in questo momento, Nonno, tu mi stia osservando e sia orgoglioso di me e del risultato ottenuto.

Table of Contents

List of Tables

List of Figures

Acronyms

DLT	Distributed Ledger Technology
dApp	distributed application
DAO	Decentralised Autonomous Organizations
DeFi	Decentralised Finance
PoW	Proof of Work
BFT	Byzantine Fault Tolerance
Pos	Proof of Stake
PoA	Proof of Authority
WFC	White Flag Consensus
L1	Layer 1

L2
: Layer 2

DMZ
: De-Militarized Zone

SC
: Smart Contract

EVM
: Ethereum Virtual Machine

ISC
: IOTA Smart Contracts

UTXO
: Unspent Transaction Output

DoS
: Denial of Service

DSA
: Digital Signature Algorithm

RSA
: Rivest, Shamir, Adleman

ECC
: Elliptic Curve Cryptography

ECDSA
: Elliptic Curve Digital Signature Algorithm

NIST
: National Institute of Standards and Technology

TCG
: Trusted Computing Group

TPM
: Trusted Platform Module

Chapter 1

Distributed Ledger Technology

Distributed Ledger Technologies have gained significant attention since the advent of Bitcoin, primarily due to two reasons. Firstly, from a financial perspective, DLT enables the transfer of value between users without the involvement of a third party, such as a bank. If transferring a small amount of money through a blockchain is inconvenient due to transaction fees, exchanging a large amount of money, such as for purchasing a house, is more cost-effective than going to a bank.

Smart contracts are the other aspect of blockchain that is gaining attention, and they were first introduced with Ethereum. Smart contracts are decentralised programs that enable us to ensure the accuracy of computations as every node in the network executes them. This feature allows for the creation of contracts between parties with the assurance that the agreed terms will be executed. Smart contracts have vast potential and can be used to implement assurance policies, *Decentralised Autonomous Organizations* (DAOs), and *Decentralised Finance* (DeFi).

As blockchain is a disruptive technology, banks initially viewed it with suspicion, dismissing it as unreliable and a scam [1, 2]. However, they are now attempting to integrate this technology into their systems [3, 4, 5].

1.1 Blockchain

1.1.1 Blockchain Introduction

The development of blockchain technology was motivated by the need to establish trust between unacquainted parties without the involvement of a central authority. The aim was to create a secure network for transactions, even among individuals who lack mutual trust. This network enables secure transactions without the need for a third party. This is a novel approach because in our society, transactions or value exchanges between untrusted parties typically require a third trusted party. For example, when purchasing a house, a notary certifies every aspect of the transaction and has the power to declare it invalid if something goes wrong.

The blockchain can be described as:

- A Distributed Ledger Techonology - DLT: a distributed database.
- Immodifiable by individual users or groups of users.
- A technology which introduces the concept of digital scarcity.

1.1.2 Distributed Ledger Techonology - DLT

Before the advent of information technology, all transaction information was recorded in a traditional book called a ledger. However, this method had some problems:

- Theft: the ledger can be stolen by anyone who can change or erase the information stored within it to his advantage.
- Human Factor Errors: when writing the ledger, it is easy to make mistakes and record incorrect information.
- Causes of force majeure: the ledger may be destroyed by unforeseen events such as flooding or fire.

The solution to all of these issues is to create copies and store them in multiple secure locations. Even now, with transactions stored in files, the problems remain similar, and the solution remains the same: creating copies. In fact, the DLT stores duplicated data among all the nodes in the network.

The advantages of the DLT are:

- Transparency and immutability: all nodes have equal access to the data and all decisions are made together. DLT provides an immutable and verifiable audit trail of all operations.
- Attack resistance: DLT is a more resilient system against cyber attacks compared to traditional centralised databases because it is distributed. This is due to the absence of a single point of failure, which makes attempts to hack such systems very costly.

The blockchain is a type of distributed ledger technology (DLT) where data is stored in linked blocks forming a chain structure. Unlike generic DLTs, which allow for four operations: Create, Retrieve, Update, and Delete; blockchains only allow for two: Create and Retrieve. The blockchain is an expanding chain of data blocks linked to each other. All network users own the entire blockchain, and they are connected to each other in a peer-to-peer network.

Pros This distributed data structure is constructed to be extremely hard to attack and to alter without shared consensus as it is nearly impossible to attack thousands or perhaps millions of users at once without leaving a trace on the chain even if every user has not so sofisticated protections. On the contrary when data are centralised all the security lies in the hands of the central authority and the security measures that it implements.

Cons When a database is public, there is no privacy, and it is possible to access all user information. However, if data is owned solely by a central authority, privacy is ensured based on the level of trust in that authority.

The privacy issue can be easily addressed through the use of cryptography. This allows for the privacy of transactions to be ensured while also enabling them to be stored in a public and fully shared database.

1.1.3 Consensus Protocol

One important issue to address is block insertion. Allowing individuals to insert blocks could result in fraudulent transactions being added in their favour. Therefore, a block insertion system is necessary to ensure the accuracy of block transactions. All blockchains require a consensus protocol, which is an algorithm that enables individuals who do not know each other and therefore have no mutual trust to agree on what constitutes a valid block to add to the blockchain, even if some users may be malicious. The first proposed algorithm was the Proof of Work.

Proof-of-work (PoW)

In order for a block to be accepted by network participants, miners must complete a proof of work which covers all of the data in the block. Due to the very low probability of successful generation, this makes it unpredictable which worker computer in the network will be able to generate the next block.[6] The Proof of Work (PoW) demonstrates that the miner has completed a specific, complex, resource-intensive task, providing evidence of their commitment and reliability.

The Byzantine Generals Problem

A modern version of this problems can be formulated as: *A number of Byzantine Generals each have a computer and want to attack the King's wi-fi by brute forcing the password, which they've learned is a certain number of characters in length. Once they stimulate the network to generate a packet, they must crack the password within a limited time to break in and erase the logs, lest they be discovered. They only have enough CPU power to crack it fast enough if a majority of them attack at the same time.*[7] This problem illustrates the need for a majority consensus to achieve an objective. To address this issue, Satoshi Nakamoto proposed a solution based on Proof of Work (PoW) and the longest chain rule. PoW represents the puzzle that must be solved to add a block, while the longest chain rule is used to establish consensus on what to agree upon. This agreement between generals enables them to launch a successful attack. The agreement on the chain and the PoW used to generate blocks form the foundation of the Bitcoin Blockchain.

Byzantine Fault Tolerance and Attack Resistance

A system that is BFT[1] is resistant to faults from up to one-third of its input sources.[8] In Bitcoin, solving the PoW puzzle results in a prize, but it requires significant resource usage. This protects the network because an attacker must have a significant amount of computational resources to solve the PoW before other users. With the current state of the network, a huge amount of computational power is required to win, at least more than the 51%[2] of the total power involved in the network, or more than 66% of the power to compromise the network's functionality and break the BFT. The network has a second layer of protection in the form of control over the proposed solution. If a miner solves the PoW, they have the right to insert a block. However, if the proposed block is deemed invalid[3] by other network users, it will be refused. This means that the miner will have wasted resources and will not be able to claim the prize.

[1]Byzantine Fault Tolerance

[2]See the 51% attack in Section 1.3

[3]An invalid signature or a double spending, for more details on double spending see Section 1.2.1

1.1.4 Proof of Stake

In this section the presented information of the Proof of Stake are taken from the Ethereum Site.[9]

In Proof of Stake, validators[4] have to put something valuable at stake into the network that can be destroyed if the validators act dishonestly. The validator is responsible for checking that new blocks propagated over the network are correct and valid, moreover, sometimes he has to create and send blocks. If the validator accepts an invalid block or proposes an incorrect block, trying to defraud the network, he will be punished with the destruction of some or all his staked ETH.

Benefits over Proof of Work

Ethereum adopted the Proof of Stake to gain this benefits over the Proof of Work:

- Energy efficiency: to solve the PoW puzzle, dedicated hardware with high energy consumption is required to compute numerous hashes to find the solution.
- Reduced hardware requirements: there is no need for elite hardware to insert new blocks.
- Economic penalties: 51%[5] style attacks are more costly for an attacker compared to proof-of-work, because there are penalties in misbehaving, while in proof of work, at least the malicious miner will see his block rejected losing his reward.
- Social recovery: if a 51% attack succeeds, the community can revert to a previous version of the network.

[4]In Ethereum a validator is a user of the network that stakes at least 32 ETH, he is in charge of proposing, voting and checking blocks. If he cheats during these phases he loses his staked funds.

[5]See Section 1.3

1.2 The Tangle: IOTA

IOTA is a distributed ledger technology that utilises the Tangle, a completely new paradigm in distributed data structures. The Tangle is a directed acyclic graph where vertices represent blocks, and each block corresponds to a transaction. Edges correspond to the validation of previous transactions. When issuing a new transaction, it must be correct and linked to two previous transactions, known as tips in this context.

Firstly, a Tip Selection Algorithm is run to choose the two tips to validate. These tips are selected based on specific policies, ensuring that they are well-formed and have a high cumulative weight. This means that the signatures inside must be valid and not part of a conflicting branch.

Afterward, an adaptive Proof of Work is executed. This PoW serves as a rate control mechanism. Without this filter, an attacker could issue a large number of transactions and potentially harm the network. Usually, when a node issues transactions, the difficulty is low. However, if the node increases the issuing rate, the difficulty also increases to lower it.

When a transaction links to another, it directly confirms it and indirectly confirms the oldest ones linked to the one directly confirmed. The more confirmations a transaction receives, the more confident the network is about its validity.

1.2.1 Consensus

The Coordinator - PoA Consensus

In IOTA, the consensus algorithm is called Proof of Authority. This name is derived from the presence of a Coordinator within the network[6]. *The Coordinator is a node plugin that sends signed blocks called Milestones that nodes trust and use to confirm blocks and reach consensus. Blocks that are directly or indirectly referenced by a milestone block are automatically deemed confirmed, though whether they mutate the ledger depends on the White-Flag consensus approach.*[10] Nodes within the IOTA network run the PoA protocol through the Coordinator and have the Coordinator's public keys saved in their configuration file to validate whether Milestones are issued by the Coordinator. The Coordinator regularly issues Milestones that determine the confirmation latency of the network.

[6]*The Coordinator is a temporary feature running under the Chrysalis and Stardust protocol versions and will be removed with the upcoming IOTA 2.0 update.*[10]

White Flag Consensus

Consensus usually consists of complex and resource-intensive computations to decide which conflicting transaction is approved but the WFC[11] simplify the process by taking advantage of Milestones. With WFC there is no need for any method to decide between conflicting blocks. It uses a deterministic method based on the appearance of transactions in the last Milestone. In other words, it gives an order to transactions. This approach of retroactively determining the order of transactions has some benefits:

- Efficiency: there is no voting phase, which makes the process quicker and requires fewer computational resources.

- Predictability: the system is deterministic, meaning that it is predictable. This allows for the ledger's state to be determined after processing transactions up to a milestone.

- Security: it is a robust mechanism for handling double spending

Double Spending

Double spending is when a user accidentally or intentionally uses the same funds in two different transactions. In Bitcoin, it means that a user tries to pay with the same address twice, while in Ethereum or IOTA, it means spending more than what the users have in their account.

To avoid this IOTA uses White Flag Consensus, it approves the first transaction that does not conflict with the ledger state, and all the others that conflict are ignored and left unconfirmed. This deterministic approach has the advantage that every node will reach the same state without exchanging any information with the other nodes.

1.2.2 Nodes

IOTA has two types of nodes, Hornet nodes that implement the Layer 1 and Wasp nodes that implement Layer 2

Layer 1 - Hornet

Hornet nodes are responsible for verifying transactions and processing client requests. They manage the state of the Tangle and run any protocols required by the chain to be active (e.g. WFC, etc.).

Layer 2 - Wasp

In IOTA, it is possible to build multiple fully ordered ledgers on top of the Tangle (L1), which are essentially L2 blockchains. On these chains, it is possible to deploy smart contracts. Wasp nodes are used to deploy L2 chains, each node can deploy and have more than one; all these chains are anchored to the Tangle by a Hornet node, and thanks to this anchoring even cross-chain transactions are possible since the Hornet can act as an intermediary and the state of each chain is stored in the Tangle.

1.3 Blockchain Security

A server responsible for verifying signatures or other values should be a highly secure machine. To ensure its trustworthiness, various security measures such as a firewall, intrusion detection system, and a DMZ (De-Militarized Zone) are applied, making the system complex. However, the downside of complexity is the increased probability of errors [12]. Alternatively, using blockchain technology ensures a secure system as it has been specifically developed to be one. The blockchain's security comes mainly from two aspects:

- *Byzantine Fault Tolerance*: the property explained in Section 1.1.3
- *the distribution*: The security of the network increases with the number of users. Each user acts as a referee, overseeing the operations of others. All users agree on the rules of the blockchain protocol. If someone attempts to cheat, the proposed modification is immediately rejected. Furthermore, each user participating in the mining process contributes to the network's power (in the case of Bitcoin), which means that the more powerful the network, the more resources (hardware in Bitcoin, ETH in Ethereum) an attacker would need to overcome the other miners/validators.

A theoretical attack on the blockchain is the 51% attack, which works like this in Bitcoin: the attacker must have a computer whose power is at least 51% of the power of the network. He takes part in the mining game by having more power than half of the network, he mines faster than the others. While mining, he makes a value transaction with another user, sending a huge amount of Bitcoin to pay for something; this transaction is inserted into a valid block mined by someone in the network, but not by the attacker. Meanwhile, the attacker, who is mining, inserts a valid block that creates a fork, this fork starts before the block containing the attacker's transaction and it is kept secret, it is not published. The miner who now has his own fork mines blocks with valid transactions and inserts these blocks into the fork, since the attacker has more power than half the network, at some point in the future his fork will be longer than the original one when this happens the attacker can publish the fork and since this is the longer chain, all new blocks will be inserted on it, in this way the original chain is abandoned and all miners will be working on the fork created by the attacker. Considering that the correct chain is now the one created by the attacker, he gains two advantages: the transaction he made is no longer valid and he has his Bitcoin back, but at the same time he has received the service he paid for, because from the moment he made the transaction to the moment the fork was published, a significant amount of time passed, so the recipient of the payment trusted that everything went well. Also, since he was the only miner on that fork, all the rewards were given to him, and probably the subsequent rewards will go to him as well since he has so much computing power.

This type of attack could be very dangerous, but given the current state of the Bitcoin network, building an infrastructure with that kind of power will be so costly that the revenue will not cover it. There are mining pools, groups of miners who come together to pool their computing power to increase the likelihood of getting the block reward and share it based on the power each user has brought to the pool. These pools are now so large that only a few of them reach the 51% of the total hash rate [13], as shown in the figure 1.1, but these groups are not interested in halting the network because that would decrease the value of Bitcoin incredibly and that is not in their interest. These groups are interested in getting as many Bitcoins as possible and keeping the value as high as possible.

As the figure shows that two very big pools alone have inserted more than half of the blocks produced in the last year, this could make think that the network is not really decentralized because most of the power is concentrated in the hands of a few actors. Anyway a pool is a set of miners and if one of them does not want to follow the pool can go away and join another or mine alone, the unknown slice represents miners that do not join any pool.

On Ethereum to affect the liveness of the chain, at least 33% of the total staked ETH on the network is required while to control the contents of future blocks, at least 51% of the total staked ETH is required, and to rewrite history, over 66% of the total stake is needed. In these cases Ethereum has some security features, the protocol would destroy these assets in the 33% or 51% attack scenarios and by social consensus (outside the network) in the 66% attack scenario. Moreover, a malicious validator can be actively punished ("slashed") and ejected from the validator set, costing a substantial amount of ETH, on the other hand in a PoW blockchain the attacker can try to repeat the attack until it has enough hash power.[14]

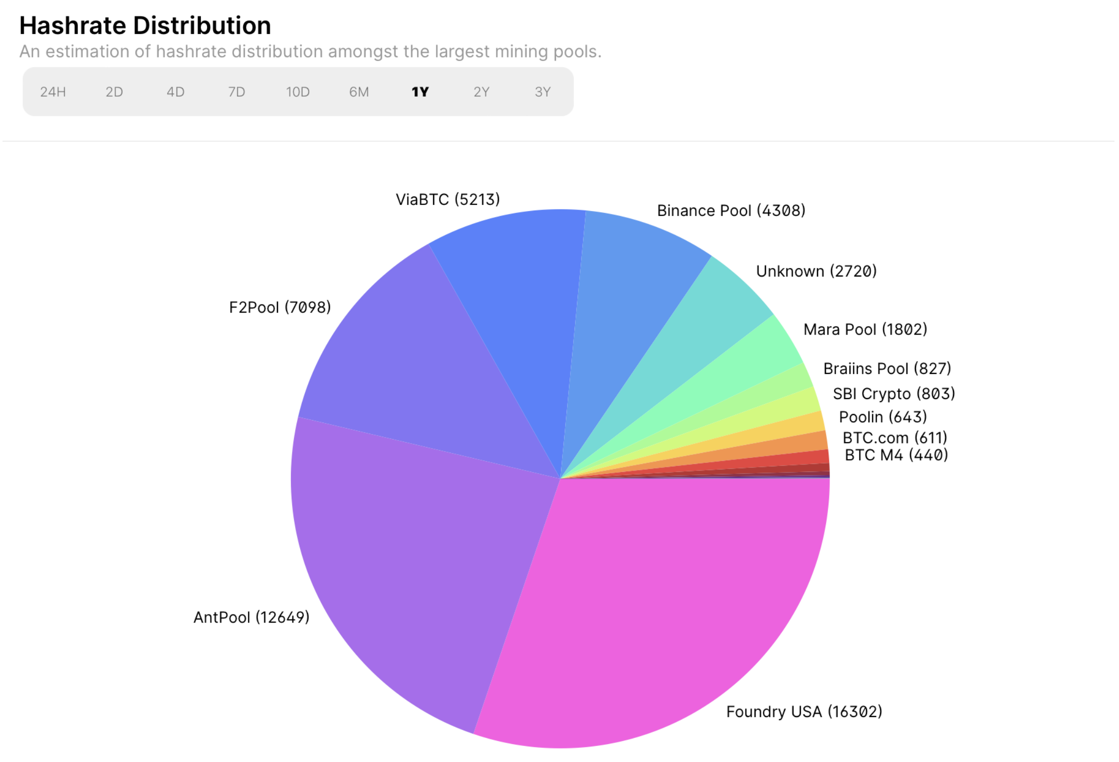


Figure 1.1: Bitcoin Mining Pools Hash Rate, data from [13]

On IOTA, to protect the network from the 34% attack, there is the Coordinator[15], a central entity that is programmed to be shut down with the Coordicide update. As IOTA is in its early stages, the coordinator only protects the network for the time needed for the technology to mature and become resilient on its own. To protect the network from the Sybil attack, IOTA introduced Mana[16], a shadow token that corresponds to the reputation of a node; if a node tries to cheat, it loses Mana, while if it is a good actor in the network, it gains Mana and increases its reputation, like in Ethereum a node that stakes a lot of ETH.

After considering the potential issues with blockchain technology, it is important to highlight its advantages. A premise is important before starting, if a developer publishes a smart contract with a security flaw, the network can not do anything to protect the users of the smart contract, like what happened in 2016 with The DAO[7]. The issues with vulnerable code exist in both distributed and centralized environments, so will not be further considered. The security advantages of a

[7]In 2016 The Dao was hacked due to vulnerabilities in its smart contracts' code.

blockchain come from its distributed nature, indeed having a large number of computers all of them executing the same code guarantees that if one or more of them fail for any reason the network can continue to work thanks to the Byzantine Fault Tolerance property of the network. This means that the network can work properly even if one-third of the network has been successfully attacked, having an infrastructure so resistant is not so easy to achieve, especially in a company where a compromised computer can lead the company to a big danger for its entire network. Another important aspect of the blockchain is its immutability, everything that has been recorded on it will remain forever[8], this ensures that any transaction done cannot be denied, for example, if payments take place, the payee cannot state that he has not received it or on the other hand the sender cannot claim to have done the payment if it is not stored inside the blockchain. If the payment has been sent to the mempool[9], it has no value because a transaction becomes effective only when it is inserted in a published block and that block receives a certain number of confirmations[10]. For example, in Bitcoin, a block is mined approximately every ten minutes. To be considered solid, a block must receive six confirmations. Therefore, when the block with the payment becomes public inside a block, the receiver must wait at least one hour to be sure the payment will not be reverted. The only way in which this could happen is through a fork, but a fork longer than 6 blocks is nearly impossible. The payee could also decide to wait for fewer confirmations, but this is a discouraged decision.

The blockchain is stored in multiple nodes, known as full nodes, which all contain an exact copy of the chain. This creates an extremely redundant environment that is resistant to multiple faults. Even if a significant number of nodes go offline, all data remains available. When the offline nodes come back online, they will synchronize with the current state of the network.

[8]Until the network will remain active.

[9]the place where all the transactions go before been chosen from a miner/validator to be inserted in a block.

[10]the number of confirmation corresponds to the number of blocks inserted after the one in consideration.

Chapter 2

Smart Contracts

Nick Szabo in 1994 wrote a paper called Smart Contract in which he gives this definition [17]:

A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitration and enforcement costs, and other transaction costs.

His idea was to theorize a new type of contract that would avoid the need for an intermediary such as a notary or, in case of dispute, a judge; this type of contract has to be applied unequivocally so that no one could disagree about its 'result' after signing it.

Then in 1996, he wrote another paper where he pointed out four design objectives that a contract should have. They are [18]:

- *Observability*: the principals should have the ability to observe each other's performance of the contract, or provide proof of their performance to other principals.
- *Verifiability*: the principal's capacity to demonstrate to an arbitrator that a contract has been executed or violated, or the arbitrator's ability to learn this through measures.

- *Privity*: the principle dictates that parties should distribute knowledge and control over the contents and performance of a contract only as far as necessary for its performance. This principle generalizes the common law principle of contract privity, which holds that third parties, except for designated arbitrators and intermediaries, should not have a say in the enforcement of a contract. Generalised privity extends beyond the common claim of ‘it’s none of your business’ to formalise it.

- *Enforceability*: a smart contract must be enforced, but enforcement should be minimized and applied only as needed. Improved verifiability can aid in meeting this objective. Reputation, built-in incentives, self-enforcing protocols, and verifiability can all contribute to meeting this objective.

Providing an example can help demonstrate the differences between traditional contracts and smart contracts. When drafting a traditional contract, the parties create an agreement that must be fulfilled if certain conditions are met. However, this approach can lead to two major problems: the need for trust between the parties involved and the enforcement of the contract. For instance, when buying a house, the buyer must trust that the seller will vacate the house, while the seller must ensure that the buyer will pay the agreed amount. Since both parties often lack trust in one another, a trusted third party is necessary to validate the contract. This third party is known as a notary, who validates their contract and in case one of them does not honour the contract, he will declare it as invalid. Having a notary involved can be pricey and sometimes not necessary in drawing up a contract. For example, a betting contract does not require a notary. When placing a bet, a bettor hands over the money to the agency but if he wins, who can assure him that he will be able to collect his winnings? The bettor has to trust the agency, or in case the agency does not honour the contract, take the matter to court and face potential expenses greater than the win. On the other hand, a smart contract could be the code of a vending machine. When a customer chooses a product, the machine displays the requirements for purchasing the item, its cost. The buyer then inserts the money, and the machine checks the conditions before dispensing the selected product. The vending machine only releases the selected product if all criteria are fulfilled. If a product is not selected or money is not enough, the machine will not dispense the product and will give back the money. A trusted third party is not necessary here because the code makes sure the machine behaves correctly, as long as the code is error-free and every slot has sufficient products. If the user selects an incorrect item, the contract still remains correctly accomplished as the mistake is due to user error and not a malfunction of the machine. The main advantages of such a contract are the automatic execution, there is no need for someone to enforce it, and predictable outcomes, when selecting an item the user is sure the machine will give exactly that one.

2.1 Ethereum Smart Contracts

Ethereum was the first blockchain technology to introduce a Turing-complete programming language that enables the creation of in-chain smart contracts.

In a blockchain environment, smart contracts are programs that operate on a distributed network with various validators[1]. By running identical code with the same inputs, they ensure the expected behaviour and prevent tampering.

However, how can we ensure that the code and inputs are identical for every validator? Thanks to the DLT every smart contract's code is published in the chain and it cannot be modified. For the input, the process is more complicated, the network, every 12 seconds, inserts a block with the inputs and the responsible for creating this block is one randomly chosen validator between a committee of validators composed randomly, whose votes are used to determine the validity of the block being proposed, in case the selected validator tries to cheat he will lose his stake.

This allows the creation of trusted applications. The code of a smart contract defines what the contract is able or not to do. The Code Is Law, this is the principle inside this technology[2].

In Ethereum, a smart contract has some characteristics[19]:

- *Computer programs*: smart contracts are de facto computer programs, they are composed of code and to be executed need to be compiled and loaded in memory.
- *Immutable*: the code is published on the blockchain and for the nature of the network once something is published it will remain forever as is. It is impossible to modify the contract code, if there is this need a new instance has to be deployed.
- *Deterministic*: The result of a smart contract's code is always the same given the same transactions and the same EVM state.

[1]A validator is a participant in a Proof of Stake (PoS) network that stakes his tokens to validate transactions and blocks and supports the security of the DLT/blockchain, for more information see Section 2.2.6.

[2]In the Ethereum network this principle was violated just one time and this created the first fork. In 2016 The Dao was hacked due to vulnerabilities in its smart contracts' code, so the community voted on what to do and decided to revert the hack creating a fork, but the ones who did not agree because The Code is Law principle continued on the hacked chain creating Ethereum Classic.

- *EVM context*: for security reasons a smart contract can access just its own state, the context of the invoking transaction, and some information about the previous blocks.

- *Decentralized world computer*: every Ethereum node has its instance of the EVM but the initial state and the transactions that modify it are the same for everyone so the final state is shared and the system works as a single 'world computer'.

2.1.1 EVM

Ethereum is not just a distributed ledger in which accounts and balances are stored, the ability to run smart contracts takes this concept to a further level from distributed ledger to distributed state machine. The concept of distributed state machine is the foundation that a smart contract needs to be executed. The Ethereum state is a comprehensive data structure containing not only accounts and balances, but also the machine state, which changes from block to block according to a pre-defined set of rules, and which can execute arbitrary machine code. The EVM [20] defines the specific rules for altering the state from one block to the next.

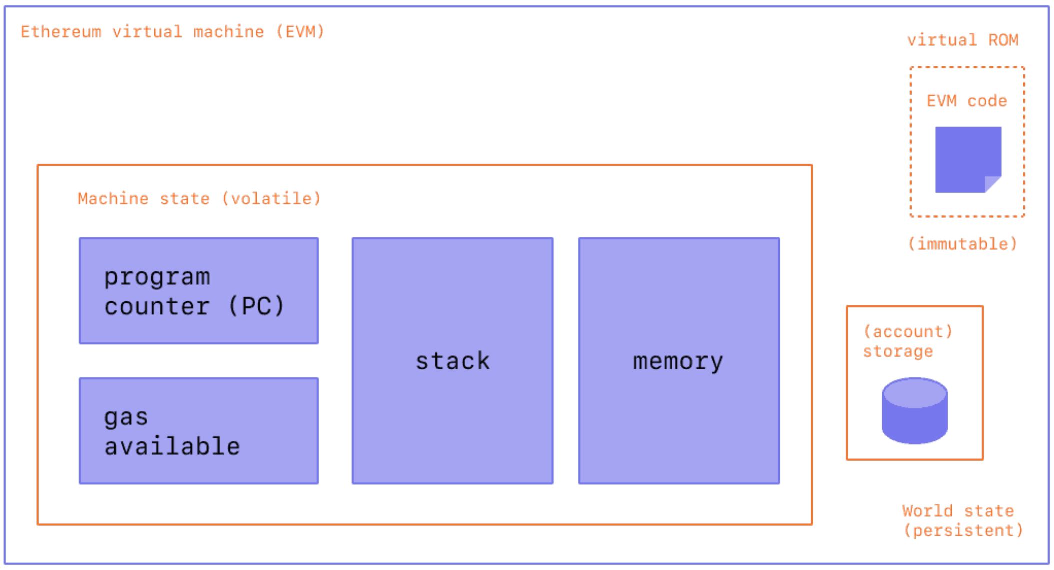


Figure 2.1: EVM [20]

The Ethereum Virtual Machine (EVM) can be seen as a mathematical function such that $Y(S,T) = S'$ [20], where starting from a valid state S and applying a set of valid transactions T a new state S' is deterministically produced.

The state is a huge data structure written in the memory as a *modified Merkle Patricia Trie*. Inside it, all accounts are linked by hashes and can be reduced to a single root hash stored on the blockchain.

The Ethereum Virtual Machine (EVM) features a stack-based design. Its stack has a 256-bit word size to optimize its Keccak-256 hash protocol and elliptic-curve computations. The memory stack size is limited to a maximum of 1024 items.

Every contract has an associated Merkle Patricia Trie connected to the smart contract account and is part of the global state.

When compiling a smart contract, a bytecode is generated consisting of EVM opcodes. These opcodes execute standard machine instructions such as XOR, AND, ADD, SUB, and more complex operations specific to blockchain and cryptography[21], such as modular arithmetic (ADDMOD, MULMOD), HASH (KECCAK256), and blockchain operations (ADDRESS, BALANCE).

2.2 IOTA Smart Contracts (ISC) Architecture

IOTA smart contracts work as a layer 2 extension of the ledger. In this layer, multiple chains can be deployed, each with its own state and smart contracts that trigger state changes. Each chain has its set of validator nodes, that execute smart contracts and record state changes. The committee agrees on a new state at each change and publishes its hash to L1. It is important to note that there is only one Tangle in layer 1. The layer 2 chain in IOTA is similar to a traditional blockchain compared to the Tangle. However, the innovation lies in the fact that L2 chains can communicate with L1 and with each other, resulting in the creation of a novel DLT protocol called ISC.

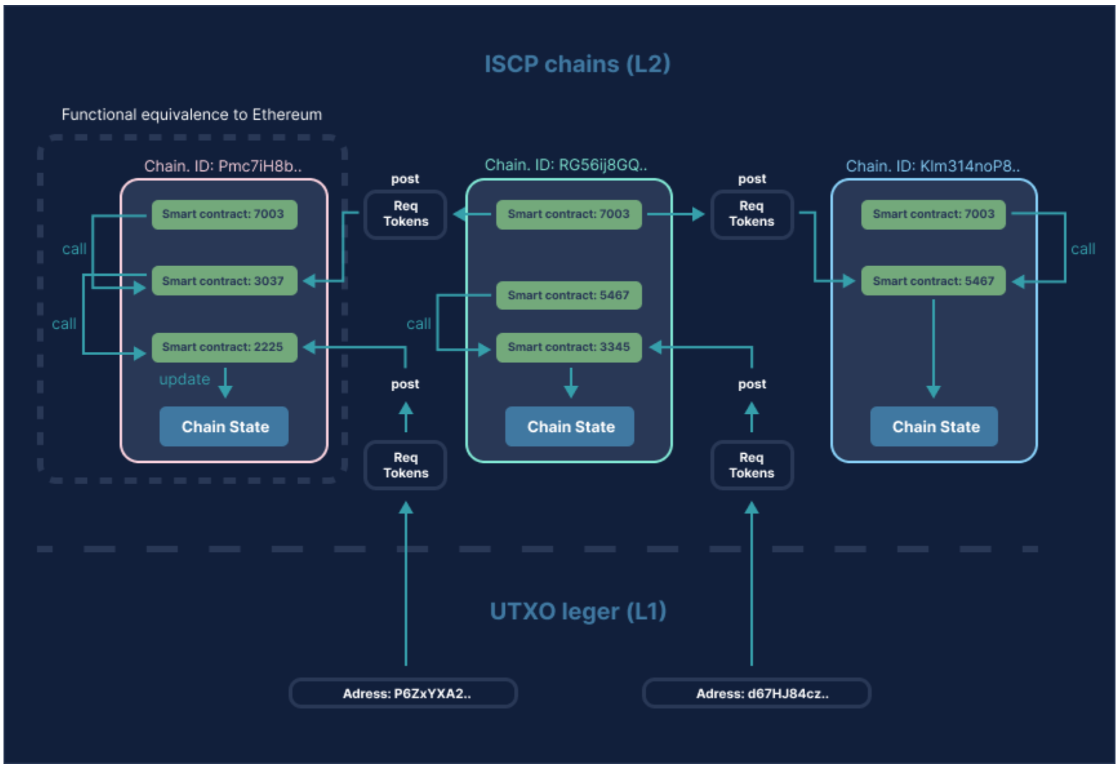


Figure 2.2: IOTA Smart Contacts multichain architecture

2.2.1 Anatomy of a Smart Contract

Smart contracts are programs that are stored immutably in the chain. However, an interpreter is required to execute them. Thanks to the *VM abstraction*, the ISC VM is independent of the interpreter used to execute each smart contract.

ISC can support different interpreters simultaneously in the same chain.
For example, WASM and Solidity smart contracts can coexist in the same chain.

Identifying a Smart Contract

Each smart contract in the chain is identified by a name called *hname*, which is a uint32 value calculated from the instance name of the smart contract; this value uniquely identifies the contract in a chain.

Smart Contract State

The smart contract state contains the data owned and stored by the smart contract and the chain, it is a collection of key/value pairs, keys and values are represented by byte arrays. The smart contract state can be seen as a private section of the chain state that only the smart contract is allowed to write to.

The smart contract has an account on the chain, which is stored as part of the chain state. This account registers the balances of the base tokens, native tokens, and NFTs controlled by the smart contract, corresponding to the funds it controls.

The smart contract is allowed to access its state and account through an interface layer called the Sandobox, only the smart contract can change data within the state and spend from the account balances. Tokens can be transferred to the account by any other party on the ledger, including an address or another smart contract.

Entry Points

A smart contract is a program with a set of entry points, which are functions that can be invoked to start the execution of the smart contract.

There are two types of entry points:

- *Full entry points*: These are functions that can change the state of the smart contract, this means that calling these functions consumes gas and the amount of gas consumed is proportional to the complexity of the executions and how many resources are needed for the execution.
- *View entry points* or *views*: These are read-only functions, they are only used to retrieve information about the state of the smart contract. As they cannot modify the state, they do not consume any gas.

Execution Results

When a request to a smart contract is executed (e.g. a call to a full entry), a *receipt* is added to the blocklog core contract. The receipt contains the details of the execution and the results of the request: if successful, the block in which it was included is ready to be added to the chain, with other information. Any event triggered by the smart contract during this execution is also included in the receipt.

Error Handling

A smart contract call can fail for many reasons (e.g. an exception) or because the conditions for the execution are not met by the caller or by the parameters provided. The caller will be charged for any gas used up to the point of failure, and the error message will be stored in the receipt.

2.2.2 Sandbox Interface

The access to the world for a smart contract has to be restricted because if a smart contract can directly read the information inside a website that is updated very frequently the execution result will change and this implies that the smart contract execution is not deterministic since at every iteration the result is different and not reproducible, so validators nodes cannot agree on the transaction.

Nonetheless, the access to in-chain data must be very careful, the owner and the developer of the contracts are not necessarily the same entity, and a malicious contract could ruin the entire chain if is able to modify all data in it, so every smart contract must be limited to its own space.

2.2.3 Calling a Smart Contract

Any smart contract is lying inside the chain waiting for some entity to interact with it, there are mainly two types of interaction:

- View entry points
- Full entry points

The view entry points allow to read the state of the contract, and the call to these methods is free of charge, it does not consume gas[3], while full entry points can perform any kind of operations that will modify the state of the contract so the caller is charged a certain amount of gas based on the complexity of the instructions

[3]further details on gas at Section 2.3.

performed, this two types of interactions could be compared to getter and setter methods, getting/reading the state is free; modifying it is possible but at a cost.

To interact with the smart contract, the sender must wrap the entry point into a transaction. The transaction must be signed with the sender's private key, and the validators will check the signature and execute the contents of the transaction. In this case, the transaction calls a smart contract method. The execution of the method can result in a new state of the chain, which will be committed to the chain.

It is important to note that when a request is received its modifications are committed only if the called method is able to reach its end without errors otherwise all the intermediate changes are reverted and the transaction is rejected, it is important to highlight that even if the execution is reverted all the intermediate operations performed are charged in gas to the caller since computational power has been used, so a best practice to follow when developing smart contracts is to perform all the needed checks at the beginning of the execution to avoid useless gas consumption.

Request

A request contains a call to a smart contract and the signature of the sender. The sender owns the funds and the assets processed within the request, i.e. the funds to pay the gas. Requests are asynchronous calls to contracts' entry points they are not executed immediately like calls[4] between contracts. Before being executed, requests need to wait until the chain's validators include them in a request batch, this implies that requests have a delay and their execution order is unpredictable.

Calls - Synchronous Composability

Synchronous composability is the name given to the ability of a smart contract after it is invoked in a request, to call entry points of another one and wait for their execution before going on. These calls are deterministic and synchronous, meaning that they will execute operations always in the same order one after another and the result is always the same. So when performing a request there is no difference if the entry point called is a unique block code or if there are many functions called inside it that can also be part of another contract, indeed it is better to structure the code in functions and contracts.

[4]calls will be treated in the next section 2.2.3.

Asynchronous Composability

Requests are not sent only from humans but also from smart contracts, for example, a user can send a request to a contract that sends a request to a decentralized exchange that will convert the user's funds from one currency to another and send them back through another request. This flow of operations is called asynchronous composability.

On-Ledger Requests

An on-ledger request is a Layer 1 transaction that validator nodes of an L2 chain retrieve from the Tangle, this transaction is validated from the Tangle that acts as an intermediary between users and the chain. It is the only way to transfer assets to a chain or between chains. It is possible to move assets between L2 chains because a wasp node can run multiple chains that are completely independent one from another.

Off-Ledger Requests

If all the needed assets for a request are already on the chain, there is no need to pass from the Tangle but it is possible to send the request directly to the chain's validator nodes, reducing the confirmation time. So off-ledger requests are preferred over on-ledger requests due to a shorter confirmation time unless the asset on which it is wanted to work is not already on that chain.

2.2.4 State, Transitions, and State Anchoring

State of the Chain

The state of an L2 chain consists of:

- A ledger of accounts owning digital assets like the IOTA base token, NFTs, and users minted tokens. The chain acts as custodian of these funds on behalf of each account owner.
- A collection of key/value pairs that represent the data used by smart contracts and stored by them on the chain.

The evolution of the chain can be expressed in formulae defining the actual state S and requests R that will be applied to the actual state, applying these requests to the state will generate a new state. $S' = R\{S\} \to S(S, R)$

The chain's state is immutable in the sense that a block can not be modified, but it is needed to append a new block to the previous one. This implies the creation of an append-only data structure maintained by the distributed consensus of its validators.

Digital Assets on the Chain

Every L2 chain has a corresponding L1 account, called *chain account*, this L1 account holds all the tokens entrusted to the chain in a single UTXO, the state output. In this way, the chain is the custodian of the assets of its users. The consolidated assets held in the chain are the total assets on-chain, which are contained in the state output of the chain. The chain account is controlled by a chain address, also known as chain ID, this is a special kind of L1 address, an *alias address*, it abstracts the controlling entity from the identity of the chain: the controlling entity of the chain may change, while the chain ID stays the same.

The Data State

The data state of a chain is a collection of key/value pairs whose values are arbitrary byte arrays. These data are stored in a permanent form outside the UTXO ledger, in a key/value form on the distributed database maintained by the validator nodes of the chain. When the state is stored in this database it is called the *solid state*. When a smart contract is executed an in-memory representation of the *solid state id* is created and it is called *virtual state*. On this key/pair collection, the smart contract can fetch its data and work on them, after the execution the *virtual state* can become solid once committed to the database if the execution is not reverted. An important property of the *virtual state* is to have many of them as candidates to be solidified, but just one of them will have the possibility to become the *solid state*

Anchoring the State

Anchoring the state means writing the hash of the Data State into the state UTXO and adding it to the L1 UTXO ledger. The UTXO ledger guarantees that there is exactly one such output for each chain on the ledger at every moment. This output is called *state output* or *state anchor*, the transaction[5] that contains the state output is called *state transaction* or *anchor transaction* of a chain. The state output is controlled by the entity running the chain, the controller can unlock or consume the state output.

[5]Transactions are the ones in the L1 while requests are the ones in L2.

With the anchoring mechanism, the UTXO ledger provides the following guarantees to the IOTA Smart Contracts chain:

- There is a global consensus on the state of the chain
- The state is immutable and tamper-proof
- The state is consistent

The state output contains:

- The identity of the chain (its L1 alias address)
- The hash of the data state
- The state index, which is incremented with each new state output

State Transitions

The data state is updated by mutations of its key/value pairs. Each mutation either sets a value for a key or deletes a key, so the associated value. Any update to the data state can be reduced to a partially ordered sequence of mutations.
A block is a collection of mutations to the data state that are going to be applied through a state transition:

$S' = R\{S\}$
$next\_data\_state = apply(currentdatastate, block)$
The state transition in the chain is atomic in an L1 transaction that consumes the previous state UTXO and produces the next one. The transaction includes the movement of the chain's assets and the update of the state hash.

At any moment in time, the data state of the chain is a result of applying the historical sequence of blocks, starting from the empty data state.

On the L1 UTXO ledger, the state's history is represented as a logical sequence (chain) of UTXOs, each holding the chain's assets in a particular state and the anchoring hash of the data state. Note that not all of the state's transition history may be available: due to practical reasons, older transactions may be pruned in a snapshot process. The ISC virtual machine (VM) computes the blocks and state outputs that anchor the state, which ensures that the state transitions are calculated deterministically and consistently.

2.2.5 State Manager

The state manager is a component that keeps the state of the node up to date retrieving new data and assuring that they will be consistently stored in the DB.

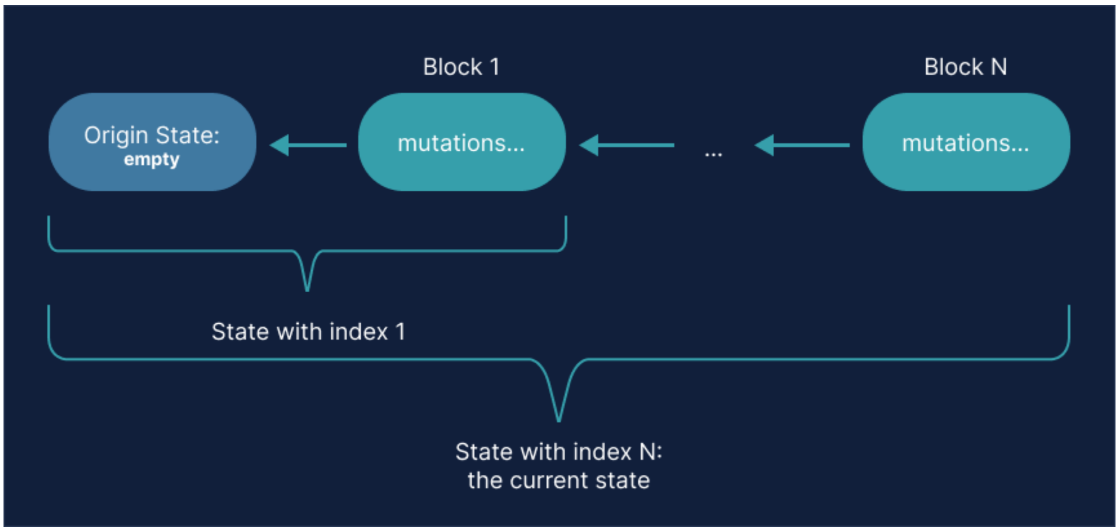


Figure 2.3: State Transitions

It answers requests from other Wasp components, these answers mainly assert that the desired state is available in the node meaning it can be retrieved from the Data State.

To obtain the last state the State Manager has to obtain the latest block, then having all the blocks from the 0 to the n it commits the transactions to the Data State in order as the blocks arrived. This approach is scalable because it does not need every time to apply all the transactions from the block 0 but if the state is updated at the block $n-2$ the state manager will only need to fetch blocks $n-1$ and n and commit their transactions into the Data State. Note that is extremely important to update the Data State following strictly the order of the block otherwise the state will not be consistent with the blocks' order and with the state in the other nodes.

There are two ways for the State Manager to obtain blocks:

1. Receive blocks directly from its node's consensus when the new state[6] is decided, note that the State Manager does not influence the consensus process.

2. Receive block from neighbours after sending them a request.

[6]In this case, state and block are used interchangeably since a block represents the variation between the new state and the precedent so that consensus on a new block implies consensus on a new state.

Snapshot

To expand the network and make it more distributed and redundant new nodes can be added. When a new node is added to the network it does not know the chain's history but it needs to have the most recent state. If the chain has been running for a while it passed through many state transitions and downloading this big amount of blocks would take a lot of time. To speed up the process, some nodes in the network, if properly configured, dump periodically a complete state of the chain into a file, this file is called Snapshot. Downloading this snapshot from a node produces the same state as downloading and committing all the blocks that lead to this state but those blocks are not downloaded and will not be available in the new node's DB, except for the block with the same state index of the snapshot.

Pruning

In order to control and limit the State DB size oldest states are pruned from it periodically. The amount of states to keep depends on two parameters: one in the configuration of the node (*stateManager.pruningMinStatesToKeep*) and one in the governance contract (*BlockKeepAmount*). The resulting limit of previous states to keep is the larger of the two. After a new state commit, the node deletes the states that are over the limit. But to avoid freezing the State Manager for too long while pruning no more than *stateManager.pruningMaxStatesToDelete* blocks are pruned in a single run. The algorithm first deletes the oldest ones to avoid gaps between available states. If desired pruning can be disabled, a node that never performs pruning and stores all the chain's state from the first one is called archive node. Note that Archive node will require a lot of resources, in particular disk storage.

2.2.6 Validators and Access Nodes

Validators

Each chain has a committee of validators who run it. The committee owns a shared key that is split between all the validators, each key alone is useless but the collective signature gives the committee complete control over the chain. The committee is in charge of executing smart contracts and committing the new status, all validators run exactly the same code and reach a consensus on the state update. After the committee computes and validates the next state, it is inserted into each validator's database and a new block is added to the chain. Finally, the state hash is saved into the L1 ledger. The chain owner has the ability to rotate the committee of validators, allowing for the addition, deletion, or replacement of validators.

Access Nodes

Some nodes can act as access nodes to the chain without the need to be part of the committee. All nodes in the chain, validators or not, are connected through statically assigned trust relationships and each node[7] is also connected to the IOTA Tangle to receive updates on the chain's L1 account. Any node can provide access to smart contracts in the chain for external callers (e.g. users) allowing them to query the state of the chain through view calls or send off-ledger requests directly to the node instead of sending an on-ledger as an L1 transaction.

It is common for validator nodes to be part of a private subnet and have only a group of access nodes exposed to the outside world, protecting the committee from external attacks. The management of validator and access nodes is done through the governance core contract.

2.2.7 Consensus

In order to update the chain, the committee must reach a consensus, requiring agreement from more than two-thirds of its validators to change the state in the same way. This approach is used to prevent any malicious nodes or errors from breaking the chain's consistency.

Smart contracts are deterministic, meaning that all nodes running the code will produce the same output when given the same input. Using the same input can pose a problem as each node may have a slightly different view of the tangle. This is due to the fact that new transactions may take time to propagate between different nodes. Additionally, validators receive smart contract requests in a random order and with a random delay, and their clocks are not synchronised. Therefore, validators must agree on the code to execute.

Batch Proposals

Firstly each node provides its vision, a *batch proposal*, this proposal contains a local time stamp, a list of unprocessed requests, and the node's partial signature of the commitment to the current state. Then the nodes must agree on which batch proposals they want to work on.

[7]Note that here chain nodes are L2-Wasp.

An example could be: there are 4 validator nodes in the committee, A, B, C and D. Each of them issues a batch Proposal. If A, B and C agree to work just on their proposals, and since they are more than two-thirds of the committee they have been able to find an *asynchronous common subset* of the batch proposals. From this point, the input is decided and will be the same for every node and everyone will be able to produce the same output independently.

The Batch

The next step is to convert the raw list of batch proposals into a definitive batch. All requests from all proposals are counted and filtered to produce the same list of requests in the same order. The partial signatures of all nodes are combined into a full signature that is then fed to a pseudo-random function that sorts the smart contract requests. Validator nodes can neither affect nor predict the final order of requests in the batch. This protects ISC from MEV attacks[8].

State Anchor

After having agreed on the input, the nodes execute every smart contract request in order, everyone producing independently the exact same block. Now each node crafts a state anchor an L1 transaction that proves the commitment to this new chain state. This transaction has a timestamp that is derived from the timestamps of all batch proposals. All nodes then sign the state anchor with their partial keys and exchange these signatures, putting together these signatures allows every node to obtain the same valid combined signature and the same valid anchor transaction, which means that any node can publish this transaction to Layer 1.

2.2.8 Core Contracts

When a new chain is deployed, seven core contracts are automatically deployed. These contracts are responsible for managing the essential functions of the chain and providing the necessary infrastructure for the other contracts that will be deployed.

root

It is the first smart contract deployed, it is in charge for initialize the chain and for create the State. It will deploy the other core contracts during the initialization process.

[8]MEV attacks, are attacks that reorder transactions in a block to gain an advantage.

accounts

This contract maintains a reliable record of on-chain accounts within its state.

blob

Its main task is to handle an on-chain registry of blobs. *A blob is a collection of named chunks of binary data [22].* Name and chunks are slices of bytes that can be arbitrarily long. Blobs are used to store data.

blocklog

The blocklog contract records requests' blocks processed by the chain and it provides views to read request status, receipts, block, and event details. The state increases its size continuously, so to save space only the last N[9] blocks are stored.

governance

It provides many functionalities:

- it defines the set of entities that will compose the state controller. It is possible to rotate the committee of validators and add or remove addresses from it.
- it defines and allows the change of the chain owner that is an L1 entity. Initially, the owner of the chain is the entity that deployed it.
- it specifies the entities allowed to own an access node.
- it defines the chain's fee policy.

errors

It has a map of error codes that correspond to error messages. This map enables the contract to store lengthy error messages only once. In case of an error, the code is returned, saving storage and gas.

[9]N parameter can be configured during the chain creation.

evm

It offers the essential framework to receive Ethereum transactions and perform EVM code. This core contract is the most important for this thesis because it allows to work with IOTA L2 chains like working with Ethereum using Solidity[10] smart contracts. Upon deployment, this contract produces a JSON-RPC that refers to the deployed chain. This must be utilised for interacting with the chain as if it were an Ethereum chain.

Entry points and Views of core contracts can be found at [23, 24, 25, 22, 26, 27, 28, 29, 30].

[10]The most popular coding language to write smart contracts in Ethereum-based blockchains.

2.3 Gas

In this section the concept of gas is explained with reference to the Ethereum network [31] in IOTA L2 Chains it works exactly the same way because of the EVM core contract as seen in 2.2.8.

Gas is essential in a distributed network capable of executing smart contracts. It serves as the fuel that enables its operation and is a vital security component to save it from coding errors in smart contracts and potential DoS attacks[11]. A poorly coded smart contract may contain an infinite loop, which could potentially block the distributed infrastructure indefinitely since it is impossible to reboot it or interrupt the execution. Gas is highly useful in this situation because for each smart contract call a certain amount of gas has to be paid, and when making the call the user has to specify the maximum amount of gas he is willing to pay for the execution if during the execution the gas ends it will raise an exception that will reverse the execution, note that the gas consumed for the execution will not be refunded. This ensures that the network will not get stuck in an infinite loop and also deters attackers from overwhelming the network with requests, as it would simply result in burning a lot of gas and potentially delaying the insertion of some blocks. This is not worth the cost of tens of thousands of euros that this kind of attack will comport. This anti-spam mechanism has significant implications for the price of gas. The payment for gas must be made in ETH, the native token of the chain. Approaching the network saturation threshold, the gas price will increase rapidly to discourage new transactions. The gas fee refers to the cost of using gas to execute an operation, which is determined by the amount of gas used and the cost per unit of gas. Gas prices are measured in Giga Wei (Gwei), where each unit is equivalent to 10^{-9} ETH, while a Wei corresponds to 10^{-18} ETH.

2.3.1 How gas fees are calculated

A user who wants to submit a transaction has to set the amount of gas he is willing to pay. The amount of gas decided is a sort of bid that will increase the probability for that transaction to be inserted in the next block by validators. If the bid is too low validators are not encouraged to place that transaction in the block since they do not earn enough, on the other hand placing a big amount will mean to surely have the transaction inserted as soon as possible but wasting ether. The user has to trade-off between insertion speed and cost.

[11]DoS: Denial of Service - an attack designed to overload a digital service with a large number of requests, causing it to stop working.

The amount of gas to pay is divided into two components: the base fee and the priority fee also called Tip. The base fee is defined by the protocol, and it is the minimum amount a user has to pay for the transaction to be considered valid. The priority fee, on the other hand, that must be added to the base fee, is a tip for validators to tempt them to insert the transaction, the higher it is, the more likely they are to choose it.

A transaction that is only willing to pay the base fee is valid but no validator will do a job for free, so there is the need to add a priority fee that is determined from the network congestion level.

The total fee formula is:

$$total\_fee = units\_of\_gas\_used \cdot (base\_fee + priority\_fee) \tag{2.1}$$

The user is able to set only the priority fee, while the protocol decides the base fee.

Base Fee

Every block has a predefined base fee, to include a transaction in that block the transaction requires a fee at least equal to the base fee. This base fee is calculated from the preceding blocks, to make this fee predictable for the users. After block creation the base fee is burned, meaning that those ETH are removed from circulation.

The base fee is calculated by comparing the gas consumption of all transactions in the previous block with a target. The base fee will increase by 12.5% per block when the target block size is exceeded. This leads to exponential growth and makes it economically unsustainable for the block size to remain high indefinitely.

It is worth noting that the occurrence of prolonged spikes of full blocks is unlikely due to the rate at which the base fee increases.

Priority Fee

The priority is an incentive for validators to include transactions in a block otherwise, they would not have any interest in fulfilling a block. A small tip assures that a validator could take into consideration inserting the transaction but does not assure the insertion of the transaction if there are others with higher tips. The priority fee is like an auction, the winners (the transactions that will be inserted) will be the ones who offer more to validators.

2.3.2 Gas Limit

The gas limit represents the uppermost amount of gas which a user is willing to pay. If the amount of gas used is lower than the maximum limit, then the rest of the gas is returned to the user. Conversely, in case that more gas is required by the execution than the limit, the transaction is reverted and the used gas will not be refunded, as the computational power has already been used. Therefore, the validator receives the gas as a reward. It is crucial not to underestimate the gas limit. Usually, transactions involving smart contracts are more costly than simple payments.

Chapter 3

Digital Signature

Please refer to [32] for the contents of this chapter, especially the mathematics.

The digital signature represents the most significant outcome of public key cryptography. It takes the form of a small piece of data that is electronically bound to a digital document, or generic file. The algorithm has two complementary implementations - one for the signer and the other for the verifier. Bob needs to send a message to Alice. While the message doesn't need to be kept secret, it is essential to ensure that Bob is the sender and that the message has not been altered during delivery. In the physical world, Bob can sign the document using his handwritten signature. However, to obtain a digital signature, Bob requires a key pair consisting of a private key and a corresponding public key. To sign a document, the message hash is calculated using a secure hash algorithm. The resulting value is then inputted into the signature algorithm[1], along with the signer's private key. The resulting signature is then attached to the message, and he is now ready to send this pair to Alice.

Alice receives the pair and needs to verify the signature to ensure the message originates from Bob and has not been altered. Initially, she calculates the message's hash using the same secure hash algorithm that Bob employed. The hash and Bob's public key[2] are then passed to the verification algorithm, which returns ok or not ok.

[1]The implementation of the signature algorithm will be addressed in the following paragraphs.

[2]Since the key is public, it can be easily retrieved from Bob.

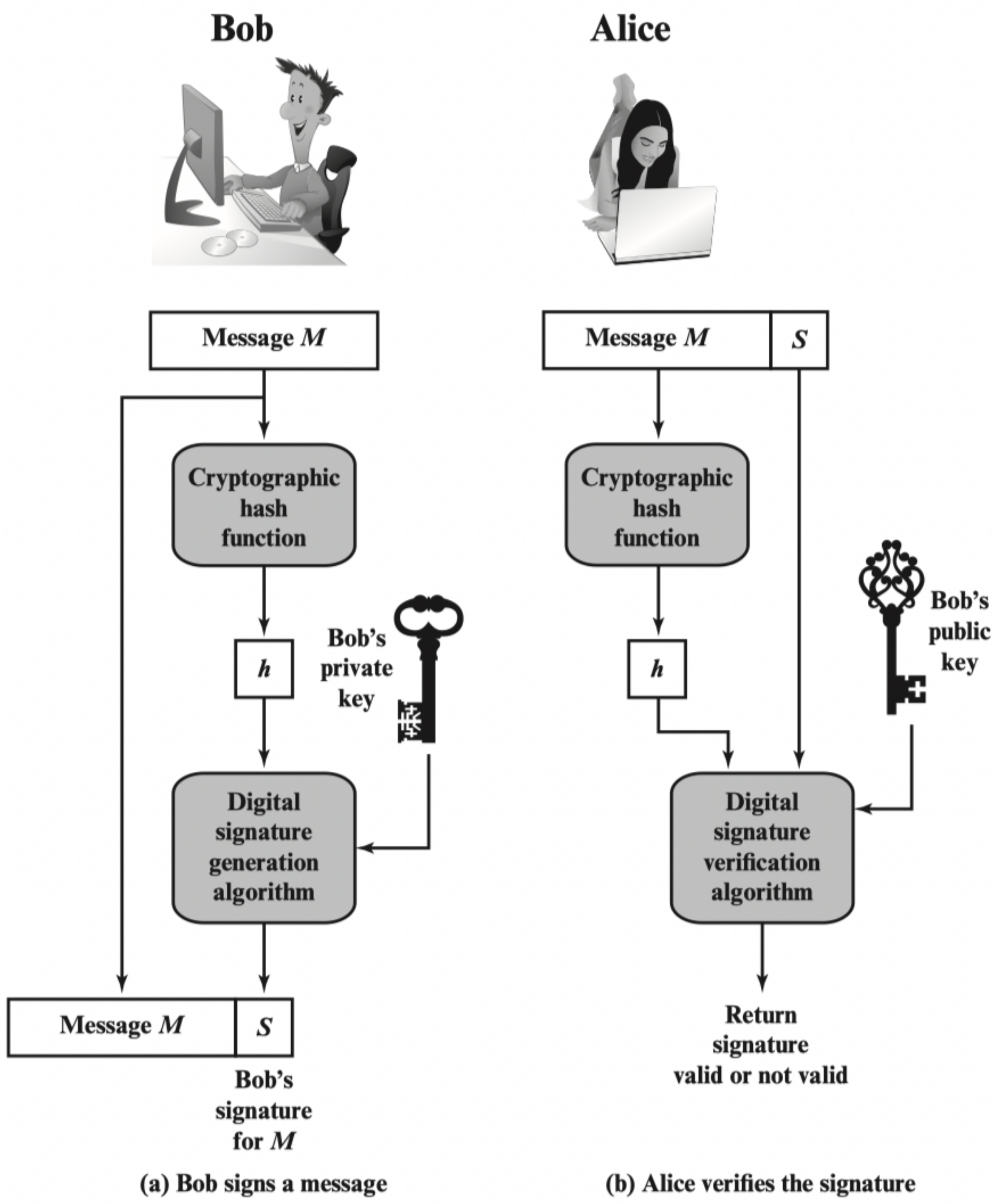

Bob
Alice
Message M
Cryptographic hash function
h
Bob's private key
Digital signature generation algorithm
Message M
S
Bob's signature for M
(a) Bob signs a message
Message M
S
Cryptographic hash function
h
Bob's public key
Digital signature verification algorithm
Return signature valid or not valid
(b) Alice verifies the signature

Figure 3.1: Simplified Depiction of Essential Elements of Digital Signature Process [32]

Digital Signature Requirements A digital signature scheme must satisfy the following rules the be considered secure:

- The signature must be a bit pattern that depends on the message being signed.
- The signature must use some information only known to the sender to prevent both forgery and denial.
- It must be relatively easy to produce the digital signature.
- It must be relatively easy to recognize and verify the digital signature.
- It must be computationally infeasible to forge a digital signature, either by constructing a new message for an existing digital signature or by constructing a fraudulent digital signature for a given message.
- It must be practical to retain a copy of the digital signature in storage.

A secure hash function is enough to satisfy these requirements.

3.1 DSA - Digital Signature Algorithm

DSA is designed to provide only Digital Signature compared to RSA which can be used for encryption and key exchange. In Figure 3.2 there is a scheme which

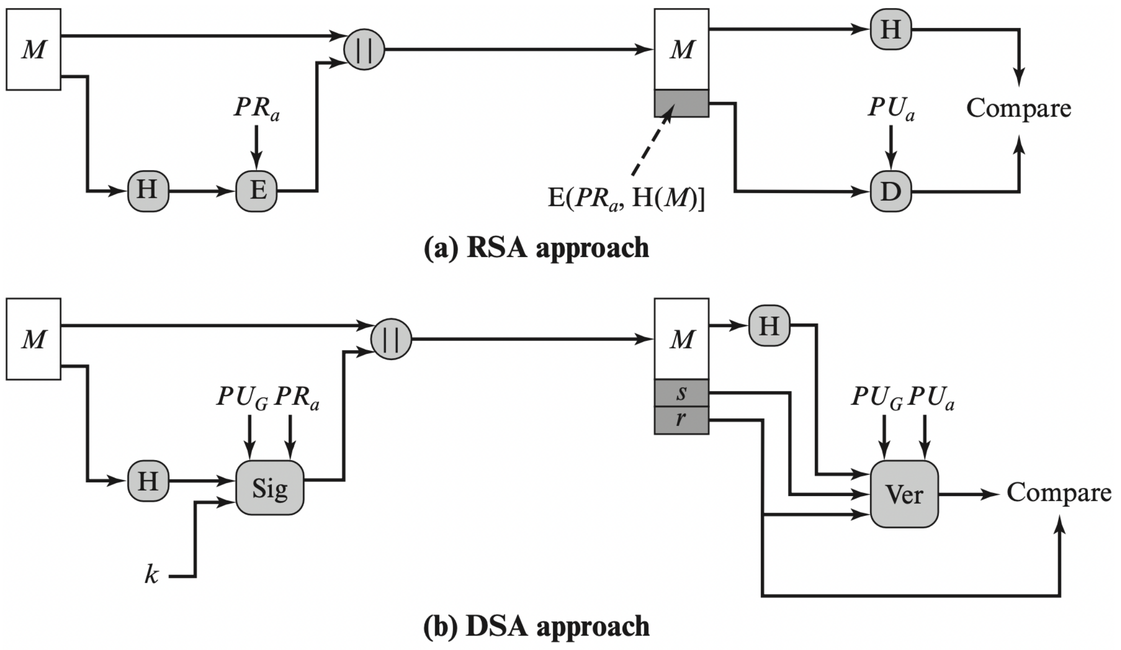


Figure 3.2: Two Approaches to Digital Signatures [32]

highlights the differences between DSA and RSA. PR_a represents a private key, PU_a is the derived public key, PU_G is the set of parameters that constitute a global public key[3]. From the figure is possible to note that in the RSA approach after calculating the hash encryption is performed on it using the private key, and for the decryption the generated signature is decrypted using the public key obtaining the encrypted hash, if the received message generates an equal hash the signature is valid. It is important to note here that this algorithm performs ciphering to sign. On the other hand in DSA there is a group of shared parameters that every user has to rely on to create a key pair. The user signs the message's hash and obtains a couple (r, s) that is the signature, the receiver to verify the signature inputs to the verification algorithm: $(r, s), PU_a, PU_G$. At the end of the process, it checks if r is equal to the computed value, if they are the same the verification is successful.

[3]It is also possible to allow these additional parameters to vary with each user so that they are part of a user's public key. In practice, it is more likely that a global public key will be used that is separate from each user's public key[32].

3.2 The DSA algorithm

The strength of this algorithm is guaranteed by the difficulty of computing the discrete logarithm and it is based on the schemes presented by Elegaml [33] and Schnorr [34].

The public parameters are three and usually are shared between groups of users. They are:

- p: a prime modulus in the interval $2^{L-1} < p < 2^L$.
- q: a prime divisor of (p – 1), in the interval $2^{N-1} < q < 2^N$.
- g: a generator of a subgroup of order q in the multiplicative group of GF(p), such that 1 < g < p. Has to be in the form: $g = h^{(p-1)/q}\ mod\ p$ with:
  - g of order $q\ mod\ p$: $g > 1$
  - $\{h \in \mathbb{N} \mid 1 < h < (p-1)\}$
- x: the private key that must remain secret; x is a randomly generated integer, such that $0 < x < q$.
- y: the public key, where $y = g^x\ mod\ p$.
- k: a secret number that is unique to each message; k is a randomly generated integer, such that $0 < k < q$.

The private key x is a *random* number in the range $[1, q-1]$. The public key is then derived from the private as: $y = g^x\ mod\ p$. This formula is the core security of this signature because it is easy to compute it but it is computationally infeasible[4] to invert it and obtain x from y, this inversion is the discrete logarithm of y to the base $g\ mod\ p$.

L	N
1024	160
2048	224
2048	256
3072	256

Table 3.1: Length in bit for L and N[36]

[4]In 2024 it is infeasible but it cannot be excluded that in the future quantum computing can break it. At the moment of writing quantum computers can factorize small integers so actually their computational power is not suitable for breaking Integer Factorization and Discrete Logarithm Problem with big numbers. In theory, it is possible, with some algorithms like Shor's algorithm [35].

3.2.1 DSA - Signature

The signature of a document is the couple (r, s) that is unique and in function of: the public key parameters (p, q, g), the message hash, and a nonce[5] k that must be chose randomly. The signing algorithm in formulae:

$$h = H(M) \tag{3.1a}$$

$$r = (g^k \ mod \ p) \ mod \ q \tag{3.1b}$$

$$s = [k^{-1}(h + xr)] \ mod \ q \tag{3.1c}$$

These formulae 3.1 are the algorithm executed inside the block Sig in Figure 3.2b. The hash result h is interpreted as a 256-bit number. The hash function H and (L, N) pair used must have a security strength that meets or exceeds the security strength required for the signature process.

3.2.2 DSA - Verification

On the other hand, the verification receives the message along with the couple (r, s). First of all, the verifier needs to know the public parameters (p, q, g) and the public key then it can start the verification following these formulae:

$$w = (s)^{-1} \ mod \ q \tag{3.2a}$$

$$h = H(M) \tag{3.2b}$$

$$u_1 = hw \ mod \ q \tag{3.2c}$$

$$u_2 = rw \ mod \ q \tag{3.2d}$$

$$v = [(g^{u_1} y^{u_2}) \ mod \ p] \ mod \ q \tag{3.2e}$$

the verification is successful if:

$$v = r \tag{3.2f}$$

These formulae 3.2 are the algorithm executed inside the block Ver in Figure 3.2b. It is important to point out that the verification strictly depends on the value of r which is independent of the message, but it is instead a function of k and the three global parameters (p, q, g) while v depends on the global parameters and h, s, r.

[5]number used once to avoid attacks

3.2.3 DSA - Mathematical Proof

$$v = ((g^{u_1} y^{u_2})\ mod\ p)\ mod\ q \tag{3.3}$$

$$v = ((g^{hw\ mod\ q}\ y^{rw\ mod\ q})\ mod\ p)\ mod\ q \tag{3.4}$$

$$v = ((g^{hw\ mod\ q}\ g^{xrw\ mod\ q})\ mod\ p)\ mod\ q \tag{3.5}$$

$$v = ((g^{hw\ mod\ q\ +\ xrw\ mod\ q})\ mod\ p)\ mod\ q \tag{3.6}$$

$$v = ((g^{(hw\ +\ xrw)\ mod\ q})\ mod\ p)\ mod\ q \tag{3.7}$$

$$v = ((g^{(h\ +\ xr)w\ mod\ q})\ mod\ p)\ mod\ q \tag{3.8}$$

$$v = (g^k\ mod\ p)\ mod\ q \tag{3.9}$$

$$v = r \tag{3.10}$$

In equation 3.4 there is just the substitution of the values of u_1 and u_2.
In equation 3.5 there is the use of properties of powers substituting the value of y.
In equations 3.6, 3.7 and 3.8 there is the use of power, sum, and mod properties.
In equation 3.9 remembering 3.1c $s = [k^{-1}(h+xr)]\ mod\ q$, it is possible to obtain the value of k, aware of $kk^{-1} = 1\ mod\ q$ and $w = s^{-1} mod\ q$, as: $k = [(h+xr)w]\ mod\ q$ that is the exponent of g in equation 3.8.
This demonstration is from [32] appendix K.

3.3 ECDSA - Elliptic Curve Digital Signature Algorithm

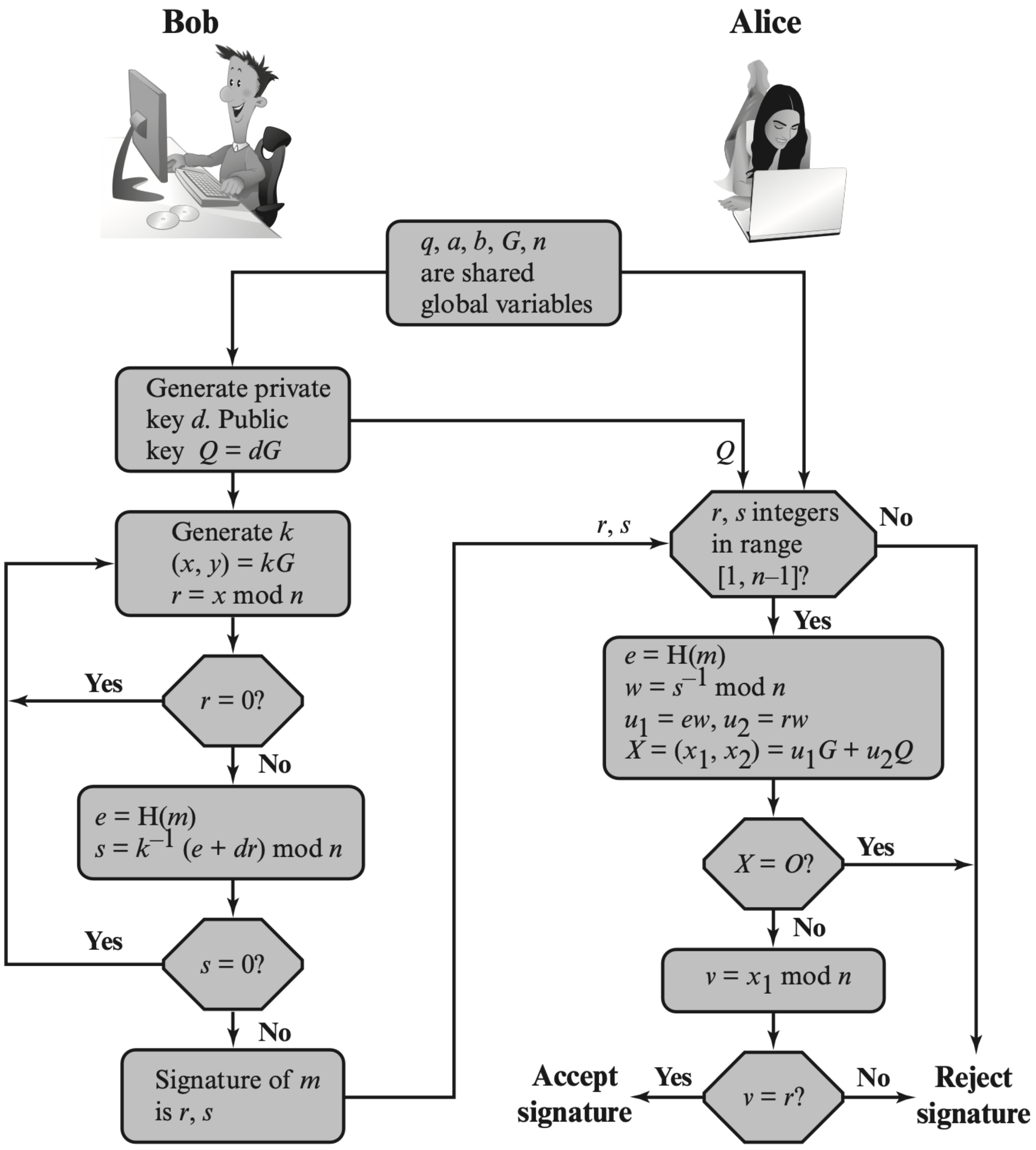


Figure 3.3: ECDSA Signing and Verifying [32]

ECDSA was first proposed in 1992 by Scott Vanstone, it was added to NIST FIPS 186-3 in July 2009 [37], but it was already used in Bitcoin, where Satoshi Nakamoto mined the first block on 3 January 2009. ECDSA has become widely used because, taking advantage of elliptic curve cryptography, it can use shorter keys than other schemes, and it offers the same security level. ECDSA is the elliptic curve analogue of DSA. As with DSA, ECDSA keys should only be used for signatures and not for any other purpose, such as key establishment. It is important to note that an adequate hash function must be used to ensure the desired level of security. ECDSA has a variant, deterministic ECDSA, suitable for devices that do not have a good source of quality random numbers. *This variant does not impact the signature verification process.*

The equation of an elliptic curve in Weierstrass' form is:

$$f(x,y): \quad y^2 - (x^3 + ax + b) = 0 \tag{3.11}$$

or making the y explicit:

$$y^2 = x^3 + ax + b \tag{3.12}$$

3.3.1 Elliptic Curve - Global Domain Parameters

In ECDSA there is a set of public parameters that define the used curve and the generator. Every user must know these parameters to use the cryptosystem.

- q: is a prime number.
- *a, b*: are the integer coefficients in the curve formula, they define the curve over $\mathbb{Z}_q$.
- This condition must be met: $(a,b) \in \mathbb{Z}_q \mid 4a^3 + 27b^2 \neq 0$
- G: is a point on the curve and it is called Generator, it is used to compute public keys.
- n: is the order of the point G. This means that n is the smallest integer such that $nG = \mathcal{O}$, which implies that n is also the number of points on the curve.
- h: curve cofactor, usually 1 in cryptography applications to avoid potential vulnerabilities.

3.3.2 Elliptic Curve - Keys

A user, to be able to sign a document, needs a key pair. To generate a key pair the user selects a random number d, such that $d \in [1,\ n-1]$. d is the secret key, with it the user can now compute the public key as: $Q = dG$. While d is a number, Q is a point on the curve.

3.3.3 Elliptic Curve - Mathematics

The computation of the public key from a private key is done in this way:

$$Q = dG = G + G + ... + G \tag{3.13}$$

On elliptic curves instead of computing multiplications a certain number of additions is done. The addition on an elliptic curve has to follow some rules:

- if three points lay on the same line their sum is the infinite point $\mathcal{O}$.

- $\mathcal{O}$ is the additive identity.
  $$P + \mathcal{O} = P \tag{3.14}$$
  and
  $$\mathcal{O} = -\mathcal{O} \tag{3.15}$$

- the negative of a point $P(x_P, y_P)$:
  $$-P = -[P(x_P, y_P)] = P(x_P, -y_P) \tag{3.16}$$
  Note:
  $$P + (-P) = P - P = \mathcal{O} \tag{3.17}$$

- To add two points with different x coordinates (if two points share the x coordinate one point is the opposite of the other e.g. P, $-P$ like in equation 3.17), draw a straight line connecting them, this line intersects the curve in a third point $-R$, the opposite of this point is the sum result R.
  $$P + Q = R \tag{3.18}$$

- In case of the addition of a point to itself, doubling a point, the line is tangent to the point and there is only another intersection $-R$ with the curve, the opposite of that intersection is the sum R.
  $$P + P = 2P = R \tag{3.19}$$

- in case the points share the same x they are connected through a vertical line and the third intersection with the curve is at the infinity point $\mathcal{O}$, accordingly with:
  $$P - P = \mathcal{O} \tag{3.20}$$

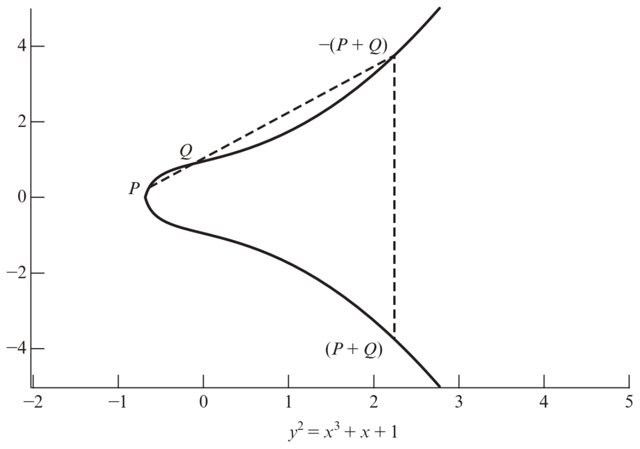


Figure 3.4: The sum of two different points [32]

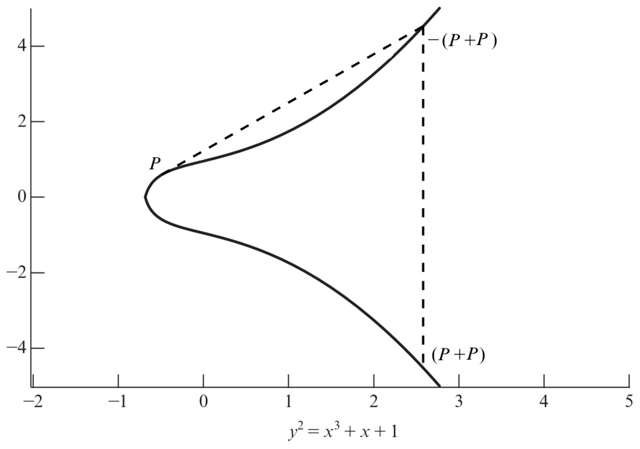


Figure 3.5: Duplication of a point, adapted from [32]

Algebraic Description of Points Addition

Defining two points $P(x_P, y_P)$ and $Q(x_Q, y_Q)$, that are not negatives of each other, the line connecting them has the slope:

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P} \tag{3.21}$$

Thanks to the preceding rules there is only one more point that intersects the line l with the slope λ. This point is the negative sum of P and Q. So $P + Q = R$ but the intersection finds $-R$.
$-R$'s x and y are[6]:

$$x_R = \lambda^2 - x_P - x_Q \tag{3.22}$$

$$y_{-R} = \lambda(x_P - x_R) - y_P \tag{3.23}$$

with those points is it possible to change sign to y_{-R} and obtain R as:

$$R = -(-R) = -(x_R, y_{-R}) = (x_R, -y_{-R}) = (x_R, y_R) \tag{3.24}$$

Algebraic Description of Point Duplication

Point duplication is calculated as

$$P + P = 2P = R \tag{3.25}$$

λ is:

$$\lambda = \frac{3x_P^2 + a}{2y_P} \tag{3.26}$$

having it, as before, it is possible to compute x_R and y_{-R} as[7]:

$$x_R = \lambda^2 - 2x_P \tag{3.27}$$

$$y_{-R} = \lambda(x_P - x_R) - y_P \tag{3.28}$$

such that:

$$-R = (\lambda^2 - 2x_P, \lambda(x_P - x_R) - y_P) \tag{3.29}$$

and in conclusion:

$$2P = -(-R) = -(x_R, y_{-R}) = (x_R, -y_{-R}) = (x_R, y_R) \tag{3.30}$$

[6]Remember that x is equal for R and $-R$

[7]Remember that x is equal for R and $-R$

Double and Add
To calculate a point on the curve quickly, it is possible to use a variant of the square-and-multiply algorithm known as *Double and Add*. Let G be a point on the elliptic curve, and suppose to calculate nG. Instead of performing $n-1$ sums, it is faster to compute $2G$, then $4G$ up to 2^kG such that $2^kG \leq nG$. Finally, the point nG can be quickly calculated.

3.3.4 ECDSA - Signature Algorithm

Knowing the public domain parameters and having a private key it is possible to sign a message, by following these steps:

1. Select a random integer k, $k \in [1, n-1]$
2. Compute the point
$$R = kG \tag{3.31}$$
and
$$r = x_R \; mod \; n \tag{3.32}$$
If $r = 0$ come back to 1.
3. Compute the message hash[8] $h = H(M)$, that is interpreted as an integer.
4. Compute
$$s = (h + rd)k^{-1} \; mod \; n \tag{3.33}$$
if $s = 0$ go back to 1.

The signature is the pair (r, s).

[8]H is a function of the SHA-2 or SHA-3 family, usually SHA-256

3.3.5 ECDSA - Verification Algorithm

To verify a signature there are some steps to follow:

1. Check that: $(r, s) \in [1, n-1]$
2. Compute: $h = H(M)$
3. Compute:
$$w = s^{-1} \ mod \ n \tag{3.34}$$
4. Compute:
$$u = hw \ mod \ n \tag{3.35}$$
and
$$v = rw \ mod \ n \tag{3.36}$$
5. Find the point:
$$Q = uG + vP \tag{3.37}$$

The signature is valid if:
$$Q_x \ mod \ n = r \tag{3.38}$$
Note that
$$w = s^{-1} \ mod \ n = (h + rd)^{-1}k \ mod \ n \tag{3.39}$$

3.3.6 ECDSA - Mathematical Proof

The verification algorithm works because of:
$$s = k^{-1}(h + rd) \ mod \ n \tag{3.40}$$
then it is possible to obtain k
$$k = s^{-1}(h + dr) \ mod \ n \tag{3.41}$$
$$k = (s^{-1}h + s^{-1}dr) \ mod \ n \tag{3.42}$$
we defined $w = s^{-1}$, so
$$k = (wh + wdr) \ mod \ n \tag{3.43}$$
$$k = (u + vd) mod n \tag{3.44}$$
in conclusion
$$uG + vP = uG + vdG = (u + vd)G = kG \tag{3.45}$$
in the last step of the verification the check over Q_x is equal to r because of 3.45 which demonstrate that $Q = uG + vP = kG$.

Chapter 4

Solidity Smart Contract ECDSA Verifier

4.1 Working Environment

The project's working environment involves a local deployment of the Iota Tangle (1.2). Three Hornets (1.2.2) nodes have already been deployed, and a Wasp (1.2.2) node needs to be deployed on top of each Hornet. The use of Wasp nodes is necessary for the deployment of the ISC chain (2.2). The process for deploying the Wasp nodes is illustrated in Appendix C. Thanks to the EVM core contract (2.2.8), the ISC chain is capable of understanding and running smart contracts written in Solidity, just like Ethereum. To generate funds needed to execute smart contracts a faucet has also been deployed.

Solidity There are many languages available for writing smart contracts, but the most popular is Solidity. Solidity is a statically typed programming language designed for developing smart contracts that run on the Ethereum Virtual Machine [38].

Javascript and Hardhat After deploying a smart contract, transactions can be sent to it using a JavaScript script through the Hardhat library. Hardhat is an Ethereum software development environment that includes various components for editing, compiling, debugging, and deploying smart contracts and dApps. These components work together to create a complete development environment [39].

4.2 Ethereum Signature Verification

To begin with, it was necessary to understand how Ethereum checks a transaction's signature and whether it is possible to reuse the code for signature verification of *generic* messages. However, using the *ecrecover* function has a major disadvantage as it only works with the elliptic curve used by Ethereum, moreover, the checked message must have the structure of a transaction, a *generic* signed message is not verifiable. The desired behaviour for the verifier is to be as general as possible, especially in implementing curves used in a TPM environment. It is important to note that the desired verifier cannot reuse any blockchain implementation. This is due to the fact that each implementation relies on just a specific curve.

4.3 Verifier Implementation

The Verifier smart contract is composed of three files: the contract defining the curves, the contract defining the mathematical operations on elliptic curves, and the Verifier. Dividing the code into parts makes it easier to handle. Note that the Curves and the ECmath contracts are not accessible from the outside, only the Verifier can call their functions.

4.3.1 Curves Contract

This module allows for the definition of all the necessary curves, which are modelled as a struct.

```

        struct Curve{
          string CurveID;
          uint p;
          uint a;
          uint b;
          uint Gx;
          uint Gy;
          uint n;
          uint h_cofactor;
        }

```

The constructor inserts all curves into a mapping[1], indexed by a *CurveID* string. To efficiently use the mapping, a vector is required to store all *CurveIDs*.

[1]Mappings are similar to hash tables [40]

This vector can be accessed by the Verifier to inform the user of all the available curves. When the Verifier needs the parameters of a curve, it can use the *getCurveParams* function of the Curve contract to obtain all the details necessary for computation inside a specific curve.

```

    function getCurveParams(string calldata CurveID)
            public view returns (Curve memory)
            {
                return curveList[CurveID];
            }

```

The constructor is responsible for populating the mapping and the *CurveID* vector during deployment.

The curves implemented and used for testing are:

- ***SECP256K1*:** this is the popular Bitcoin curve used also in Ethereum. [41, 42, 43]
- ***ECC_NIST_P256*:** this is a standard NIST curve, this is also a standard TCG (Trusted Computing Group) TPM curve. [44, 45]
- ***TPM_ECC_BN256*:** this is a standard TCG TPM curve. [45]

4.3.2 Verifier Contract

This contract is the one that receives transactions from outside the blockchain, it exposes three functions:

- *function getAvaibleCurves() external view returns(string [ ] memory)*
- *function getCurve(string calldata ID) external*[2]
- *function validateSignature(bytes calldata m, uint256 r, uint256 s, uint256 pubKx, uint256 pubKy) external*

The *getAvaibleCurves()* is used to inform a user about the available curves to check if the one that the user wants to use is implemented. This function does not consume gas since it only reads the state.

[2]The *getCurve(...)* is accessible from the outside because it was used in a test environment, but before using it in this way pay attention to Section 4.3.2

The *getCurve(string calldata ID)* on the other hand is critical because it sets the curve on which to validate the signature, considerations on its safe use are presented in Section 4.3.2. The *validateSignature(. . .)* is the function that checks if a signature is valid or not and in both cases it emits a dedicated event. It receives as parameters the message, the signer's public key and the signature. The implementation of this function is exactly the Solidity implementation of the formulae presented in Section 3.3.5[3].

Together with these external functions, there are two internal functions used inside the *validateSignature(. . .)* for computing the verification, these functions are:

- *function addPoints(uint256 x1, uint256 y1, uint256 x2, uint256 y2) internal view returns(uint256 x, uint256 y)*
- *function scalarMul(uint256 s, uint256 xp, uint256 yp) internal view returns(uint256 x, uint256 y)*

Both these functions at the beginning check that the input points are on the curve, and at the end the functions perform the same check for the obtained point.

All the operations performed in these two functions use functions of the EC math contract, which is where the mathematical part is implemented. These functions are needed for the checks and also because the functions inside the EC math require always the parameters of the curve and it leads to calls with many parameters that are prone to errors, that instead are wrapped inside these two functions that only needs the operands for computing the result, since once the curve is set the parameters are available for any function inside the Verifier.

Critical Aspects

The smart contract can operate on multiple curves. However, selecting a curve is a critical operation because the blockchain is an entirely asynchronous environment. It is not possible to call both *getCurve* and *validateSignature* in two different transactions. The reason for this is that the two transactions can be executed in any order, or worse, another call to *getCurve* could occur before the *validateSignature*, resulting in a change of the curve and an incorrect outcome.

The safest and simplest solution is to include the *getCurve* inside the *validateSignature* function. This way, during verification, the curve is also selected. Since each transaction is an atomic entity, it is impossible to change the curve during the verification. This solution was implemented to address the issue of the curve changing before signature verification.

[3]The hash algorithm used is SHA256

Another solution could be to not use anymore a Curve Contract but hard code the curve parameters in the Verifier contract for only a curve and have multiple Verifier deployed, each one implementing its curve, but duplicating the code makes it harder to maintain [46, 47], and on the contrary, it is strongly discouraged.

4.3.3 EC Math Contract

This is the contract where all the mathematical operations on elliptic curves are implemented.

The functions that are used by the Verifier are:

- *function invMod(uint256 x, uint256 pp) internal pure returns (uint256)*
- *function isOnCurve(uint x, uint y, uint aa, uint bb, uint pp) internal pure returns (bool)*
- *function ecAdd(uint256 x1, uint256 y1, uint256 x2, uint256 y2, uint256 aa, uint256 pp) internal pure returns (uint256, uint256)*
- *function ecSub(uint256 x1 uint256 y1, uint256 x2, uint256 y2, uint256 aa, uint256 pp) internal pure returns (uint256, uint256)*

These are the functions needed to compute the verification of a signature.

4.3.4 Implementation

Affine Coordinates

The Verifier needs to perform computations on elliptic curves and to do that it mainly uses functions from the EC Math contract. Initially, the EC Math implementation uses affine coordinates, but this leads to overflow errors due to the points being integers on 256 bits. Solidity can only handle numbers up to 256 bits, so scalar multiplication between large numbers leads to this error. Another common error is subtracting a larger number from a smaller one, resulting in a negative integer. While Solidity can handle negative numbers, in this case, every variable is an unsigned integer and the leftmost bit is not used to indicate the sign. Therefore, if signed numbers were used, the range of numbers that could be handled would be even smaller.

These errors appear with equation 3.22, λ^2 produce an overflow that can be solved using the Solidity function *mulmod()* this function does not suffer overflow and gives back $\lambda^2 \ mod \ p$ but this number is smaller than $x_P + x_Q$ that when subtracted to $\lambda^2 \ mod \ p$ generate a negative number.

To overcome this error the first attempt was to enable Solidity by using integers bigger than 256 bits, in this way the subtraction in equation 3.22 does not generate a negative number because λ^2 is not taken in modulus. It is possible to use the library BigNumber [48], the use of this library makes possible to compute numbers bigger than 256-bit, so it is possible to compute the sum of two points and the scalar multiplication.

Note that also equation 3.21 can generate errors, in case the subtractions give a negative number. Also for this case, the BigNumber library is useful since there is no problem in handling the sign of a number, so it is possible to compute the modulus of a negative number and then have the correct numerator and denominator for the λ computation.

The use of this library requires the conversion from BigNumbers to uint multiple times in the verification algorithm, such as after summing two points, the coordinates need to be returned as uint256. This process can be quite expensive as it involves analyzing the BigNumber as a string of bytes and extracting a number from it. However, in this case, this manipulation will allow the contract to work as the BigNumber is only used for intermediate steps and the results are always in 256-bit thanks to the modulus.

Affine Coordinates - Problems

The use of affine coordinates simplifies the writing of the algorithm, but requires the use of a BigNumber library for computation and conversion of BigNumbers into a standard Solidity type (*uint*). However, this process has been found to consume a large amount of gas when handling small test numbers and causes the transaction to run out of gas when using numbers with an order of 10^5. The implementation is unsuitable for a distributed environment like the blockchain due to its extreme resource-intensiveness. The high gas consumption is caused by the use of a BigNumber data structure to store numbers, which takes up a significant amount of bytes and requires extensive processing. In Ethereum, gas consumption is proportional to the complexity of the operation, the number and type of assembler operations required by a function, and the number of bytes the function processes. The verification algorithm in this case involves many complex and expensive mathematical operations. Additionally, the data on which these computations operate are large, requiring numerous conversions to transform BigNumbers to *uint*. The verification on a smart contract may be unfeasible unless a significant optimization is performed.

Jacobian Coordinates

Jacobian coordinates [49] are an alternative way of representing a point on elliptic curves. Jacobian coordinates correspond to an isomorphism between curve $E(a,b)$ and curve $E(aZ^2, bZ^4)$. In Jacobian coordinates a point is defined as (X,Y,Z) a triple, where X,Y,Z are coordinates related to affine coordinates x and y by:

$$x = X/Z^2 \tag{4.1}$$

$$y = Y/Z^2 \tag{4.2}$$

The curve in Jacobian coordinated is given by:

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{4.3}$$

where a and b are the same coefficients of the form in affine coordinates, equation 3.12. The Jacobian coordinates of the point at infinity are $\mathcal{O} = (0{,}1{,}0)$

In Jacobian coordinates the formulae for addition, point doubling and point multiplication are different. These formulae are complete: they work properly for all inputs, including the neutral element $\mathcal{O}$

Point Addition Consider two points $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2, Z_2)$, $P + Q = R = (X_3, Y_3, Z_3)$, to obtain R these steps are needed:

$$S = (Y_2 \cdot Z_1^3 - Y_1) \cdot (X_2 \cdot Z_1^2 - X_1) \tag{4.4}$$

$$M = (X_1 \cdot Z_2^2 - X_2) \cdot (Y_1 \cdot Z_2^3 - Y_2) \tag{4.5}$$

$$U = Z_1 \cdot Z_2 \cdot (X_2 \cdot Z_1^2 - X_1) \cdot (X_1 \cdot Z_2^2 - X_2) \tag{4.6}$$

$$X_3 = S^2 - U - 2 \cdot M \tag{4.7}$$

$$Y_3 = S \cdot (M - X_3) - 8 \cdot Y_1^2 \cdot U^2 \tag{4.8}$$

$$Z_3 = (Y_1 + Z_1)^2 - Y_1^2 - Z_1^2 \cdot U \tag{4.9}$$

Point Doubling Consider a point $P = (X, Y, Z)$, its doubling is $2P = R = (X_2, Y_2, Z_2)$, to obtain R these steps are needed:

$$S = 4 \cdot X \cdot Y^2 \tag{4.10}$$

$$M = 3 \cdot X^2 + a \cdot Z^4 \tag{4.11}$$

$$U = 2 \cdot Y \cdot Z \tag{4.12}$$

$$X_R = S^2 - 2 \cdot U \tag{4.13}$$

$$Y_R = S \cdot (U - X_R) - 8 \cdot Y^4 \tag{4.14}$$

$$Z_R = 2 \cdot Y \cdot Z \tag{4.15}$$

Point Multiplication The multiplication of a point with itself is done by adding it repeatedly and can be efficiently done using the double-and-add algorithm.

Advantages Using Jacobian coordinates resolves the problems introduced in 4.3.4, and moreover has various optimizations like:

- point addition involves fewer operations compared to the formulae in affine coordinates.
- Affine coordinates require inversion operations (division) when performing point additions, in contrast, Jacobian coordinates can avoid these inversion operations, leading to faster computations.

To take advantage of the Jacobian coordinates a Solidity library called elliptic-curve-Solidity [50] fits very well to be implemented in the EC Math contract 4.3.3, with just some minor modifications to work with the other contracts. The implementation of this library inside the EC Math contract allows now to the verifier to work with 256-bits numbers without the need for BigNumbers representations or conversion that can slow down the process.

4.4 Correctness Verification

The tests that will be presented here have been conducted in a way that the contract was not accessible from external users, without sharing the contract's address in this way the critical aspects presented in section 4.3.2 are not a concern and the solution proposed as implemented was not still used here to allow to test the functions getCurve and validateSignature as separate entities to spot eventually present problems faster. This is mainly because Solidity does not have a good step-by-step debugger so it is better to test everything separately if possible.

Being sure about the correct implementation of the signature verification is a crucial part of the verifier. To achieve this result the NIST makes available test vectors to prove the correctness of many cryptography algorithms, in this case, the needed test vectors are the ones for *ECDSA Signature Generation Component* [51] and can be downloaded at [52]. These test vectors have values computed with different elliptic curves and for each curve many SHA algorithms are present; in particular, the available curves are:
P-192, P-224, P-256 P-384, P-521, K-163, K-233, K-283, K-409, K-571, B-163, B-233, B-283, B-409, B-571,
and for each curve, the hash of a message is computed with these algorithms:
SHA-1, SHA-224, SHA-256, SHA-384, SHA-512.
For each couple curve-algorithm, in the test vectors, are present 7 fields:

- Msg: the hash of a message computed with a specified hash algorithm.
- d: the private key
- Qx: the x component of the Public Key of d
- Qy: the y component of the Public Key of d
- k: nonce
- R: r field of the signature
- S: s field of the signature

Between the curves in the test vectors, there is the curve P-256 that is also implemented in the Curve contract, and it is the only one in common, for example, the Bitcoin curve Secp256k1 is NIST non-recommended. But according to [53] in the ECDSA section it says:

"To validate an implementation of ECDSA, the algorithm implementation must implement at least one NIST-recommended curve. It can have non-recommended NIST curves if there is at least one NIST-recommended curve."

So it is enough to validate the verification algorithm just using the P-256 curve, and if it works properly with it, it works properly also with other curves. Using these test vectors implies a minor modification to the validation code because the MSG field is the hash of a message and not the message itself, so when testing the smart contract it is necessary to remove the hash computation of the message and update the function's signature.

This modification allows the use of NIST test vectors to validate the algorithm. To check the correctness of the code two kinds of approaches can be followed: the first one, to check if a correct message is recognized as such which is performed with test vectors, and the second one, to check if an incorrect message is recognized as being so.

After succeeding with test vectors to test if a correct message is recognised, a modification of the testing script is necessary to test if a wrong message is rejected and also in this case the smart contract succeeds too.

4.4.1 Testing Script

The interacting script is an easy javascript program, written with the Hardhat library that reads the vectors and performs a transaction calling the smart contract's function *validateSignature()* and feeds it with values read from the vector. These values are all interpreted as a number, except the hash, so to test a wrong signature is easy because it is just necessary to increment of 1 r or s to have a signature non-corresponding to the hash.

It is important to note that handling integers with a length of 256 bits in JavaScript is only possible with the BigInt object. A proper initialization is crucial, as the input must be a string representing a number in hexadecimal form starting with the prefix '0x'. Otherwise, it will be interpreted as a base JavaScript number and saturated to 2^{53}, which will break the verification.

4.5 Cost Evaluation

To estimate the cost of a single verification, a specific script designed for this purpose is required. When a transaction is sent, the Blockchain responds with its result. The contract's response can be of two types. The first type is when the called function ends with a *return* and the script receives what the *return* sends back, but this type of interaction is not appropriate when communicating with outside of the DLT, even if it is valid. In Solidity, a function should use the *return* when this function is called inside the blockchain by other functions of the same or another contract. In this case, only the returned values are received, without any additional information such as the block in which the transaction is inserted or the amount of gas consumed.

To obtain this further information, the smart contract needs to implement *Events*. An *Event* is a structured way for the Smart Contract to communicate with the caller. It has a self-explanatory name and can also carry parameters useful for the receiver. The Verifier contract implements three *Events*:

```

    event CurveSet(string curveID);
    event CorrectSignature(bool returnValue);
    event WrongSignature(bool returnValue);

```

Each of these *Events* has a meaningful name and brings also a parameter, the CurveSet has a parameter that informs about the id of the just settled curve, while CorrectSignature and WrongSignature [4] carry a boolean useful to understand inside an *if* if the verification was successful or not.

To catch an *Event* and all the other information related to the transaction the script must use this structure:

```

    let tx = await contract.validateSignature(Msg, R, S, X, Y);
    let recepit = await tx.wait();
    let event = recepit.logs[0];
    console.log(event.eventName);
    console.log(event.args[0]);

```

This part of the script shows the call to a smart contract function then it waits for the receipt. Inside the receipt a lot of information is available. It is composed of various fields[54]:

- *Block Hash*: String, 32 Bytes - hash of the block where this transaction is in.
- *Block Number*: Number - block number where this transaction is in.
- *transactionHash*: String, 32 Bytes - hash of the transaction.
- *transactionIndex*: Number - integer of the transactions index position in the block.
- *from*: String, 20 Bytes - address of the sender.

[4]CorrectSignature brings always True while WrongSignature False

- *to*: String, 20 Bytes - address of the receiver. null when it is a contract creation transaction.
- *cumulativeGasUsed*: Number - The total amount of gas consumed in the block where the transaction is inserted.
- *gasUsed*: Number - The amount of gas used by this specific transaction alone.
- *contractAddress*: String, 20 Bytes - The contract address created, if the transaction is a contract creation, otherwise null.
- *logs*: Array - Array of log objects, which this transaction generated.
- *status*: String - '0x0' indicates transaction failure , '0x1' indicates transaction succeeded.

Inside logs in this case the first and only element is the event generated from the smart contract. In turn, the event is an object with a name and an array of arguments, the ones defined inside the smart contract. The two console logs print the event name and the first (and only) argument of the event. For the cost evaluation, the event allows to understand the outcome of the transaction so it is important to receive always correct signature events. However, for the cost analysis, the most valuable field is the *gasUsed.* This field represents the amount of gas used to execute the called function even if it does not represent the real price that the caller will pay. It is what the caller has consumed and has to pay following the formula 2.1. The formula highlights clearly that the price to pay depends on the gas used and the gas price that has two parameters, base fee and priority fee fluctuates in time but the *gasUsed* does not, it depends exclusively on the operations inside a function, and if a function does always the same things the *gasUsed* will not change. The cost evaluation of gas will use the parameter *gasUsed* to avoid variations over time in the total fee.

For each of the following tests, every smart contract has deployed in three different chains:

- *Sepolia*: one of the most popular Ethereum test chains available.
- *Shimmer Test Network*: this is the IOTA L2 test chain implementing the Shimmer protocol.
- *Local Chain*: a local deployment as explained in Section 4.1.

Every call to a contract's function has been repeated 100 times and the mean is taken.

4.5.1 Comparison between Verificator and other common operations

In this case as in Section 4.4 the cost evaluation has been conducted keeping separate the functions ***getCurve*** *and* ***validateSignature*** *to evaluate the cost of these functions separately. During this analysis, the* ***validateSignature*** *will have two variants, the M and the H, in the M the Verifier receives the message and has to compute the hash of it while the H already receives the message hash.The choice to test these two version is to evaluate the impact of computing the hash in the signature evaluation process. Moreover, the H version is the one used with NIST test vectors for correctness verification as explained in Section 4.4.*

First of all, it is useful to compare the cost of the signature verification with other common operations in the Ethereum environment.

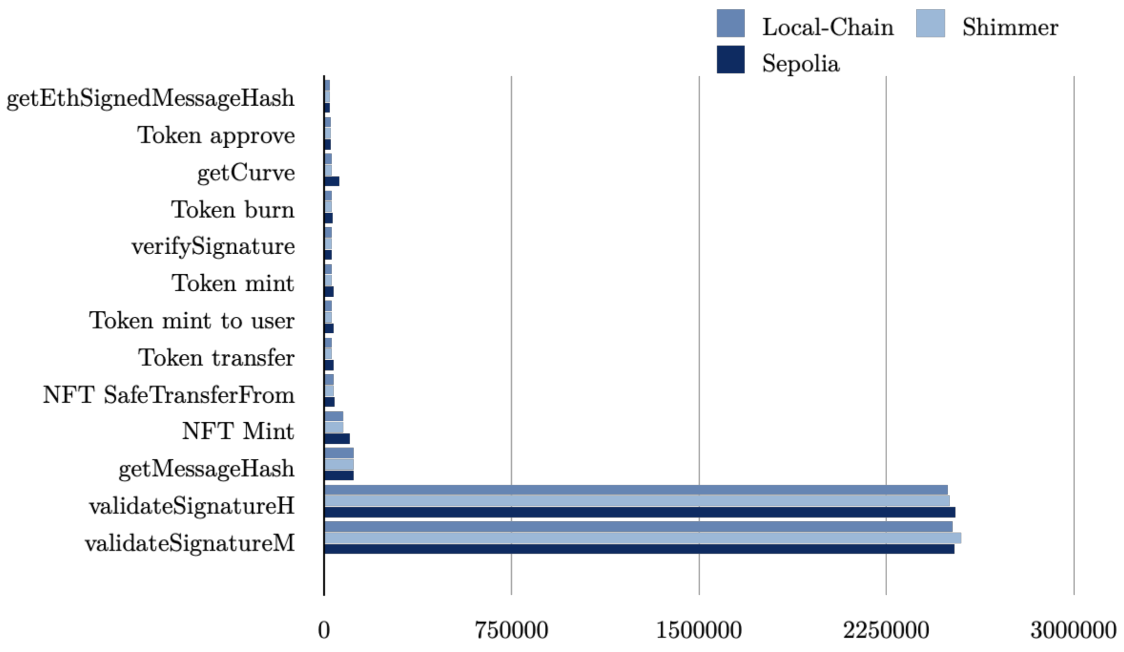


Figure 4.1: Cost Comparisons between the Verifier contract's functions and other common functions

The Chart 4.1 displays the gas cost for various functions, with each function having three lines representing each testchain.

The functions that are present in the chart belong to 4 contracts:

- *Token contract:* this is the standard implementation of a smart contract that handles tokens, it can mint, burn and transfer tokens.
  - *Token mint*: this function generates new tokens, the owner of these tokens is the contract's owner.
  - *Token mint to user*: this function generates new tokens but the owner is different from the contract's owner and should be specified when calling it.
  - *Token transfer*: this function transfers a specified amount of tokens from the caller to another address.
  - *Token approve*: this function enables an address to transfer tokens from the caller's balance to another address. This function is particularly useful when trading tokens because the seller wants to wait for the funds before sending tokens and the buyer wants the tokens before sending funds. To overcome this situation a third party, a smart contract, gets approval from both and fairly performs the exchange without cheating. It has the allowance of the right amount from both addresses so it can send to the other one the correct amount, and if one of them does not approve the correct amount the contract will not perform any exchange.
  - *Token burn*: this is the function to remove from the circulation a certain amount of tokens.
- *NFT contract*: this is the standard implementation of a smart contract that handles NFTs, it can mint and transfer NFTs.
  - *NFT mint*: this function generates new NFTs, the owner of these NFTs is the contract's owner.
  - *NFT safeTransferFrom*: Safely transfers token from the owner to a new address, checking that the receiver is (if it is a smart contract) aware of the ERC-721[5] standard to prevent NFTs from being forever locked.

[5]ERC-721 is a free, open standard that describes how to build non-fungible or unique tokens on the Ethereum blockchain, ERC-721 tokens are all unique. [55]

- *OpenZeppelin verifier*: this contract uses the standard implementation from OpenZeppelin to sign and verify a transaction on Ethereum.
  - *getMessageHash*: This function computes the hash a of message, mainly used to hash a transaction message, then the message will be signed.
  - *getEthSignedMessageHash*: this function hashes the hash of a signed transaction message but in a specific format. The digest is calculated by prefixing to a bytes32 *messageHash*: `"\x19Ethereum Signed Message:\n32"` and hashing the result. The input to this function is a message of 4096 characters.
  - *verifySignature*: this function checks the signature of a transaction, it receives as input: the signer address, the output of *getEthSignedMessageHash* and the signature.
- *Verifier contract*: this is the contract developed in this thesis.
  - *getCurve*: This function receives a curve ID and then sets that curve as the curve to use for the signature verification
  - *validateSignatureH*: This is the function that receives the signature and the hash of a signed message, not the message itself.
  - *validateSignatureM*: This is the function that receives the signature and the signed message. The input to this function is a message of 1024 characters.

As it is possible to see from Figure 4.1 on every test chain the cost is similar for the same operation. The most expensive functions are the *validateSignatureM* and *validateSignatureH.* This is because compared to the other functions they involve a lot of heavy mathematical computations. These two functions can be compared for functionality to the *verifySignature* from the OpenZeppelin contract but there is a deep difference. The Verifier contract implements in software all the logic while the *OpenZeppellin verifySignature* takes advantage of the *ecrecover(hash, v, r, s)* [56] function that is built-in Solidity and so cheaper.

4.5.2 Message length impact on the cost

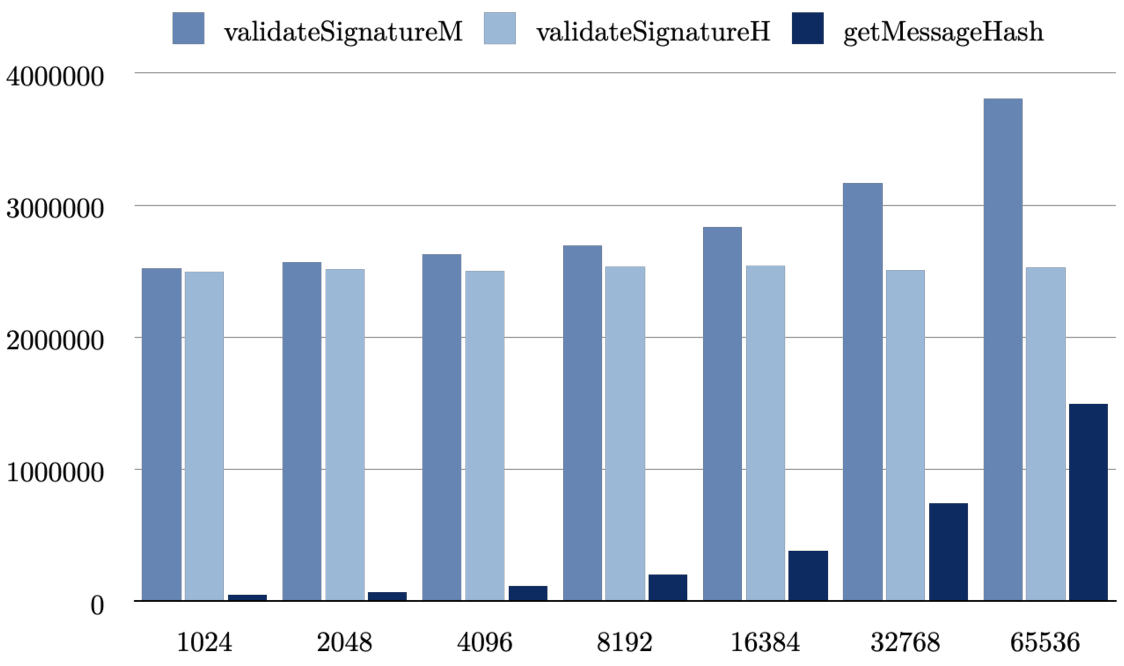


Figure 4.2: Cost Comparisons between Verifier functions increasing the message length

Figure 4.2 shows the impact of incrementing the message length on the evaluation of a signature with the methods *validateSignatureM* and *validateSignatureH* and it is quite evident that the message length has an impact on *validateSignatureM* because this function receives the entire message and has to compute the hash of it. On the other hand, the *validateSignatureH* has a constant cost independently from the message length because it already receives the hash and does not need to compute it again. This chart also displays the cost of the function *OpenZeppellin getMessageHash* when variating the message length and it is possible to see that follows the same increment of the *validateSignatureM*. Figure 4.3 shows better the correlation between *validateSignatureM* and *getMessageHash* because in the chart it is printed the difference $validateSignatureM - validateSignatureH$ and the *getMessageHash*. Looking at this image it is clear that the higher cost of *validateSignatureM* over *validateSignatureH* is exactly the cost of computing the hash, moreover the implementation inside the *validateSignatureM* is cheaper than the one in *getMessageHash* from OpenZeppellin. These results are obtained in the local deployment of the ISC L2 but the same results are obtained from tests in Shimmer and in Sepolia, the charts for those tests are available in Appendix D Figure D.5.

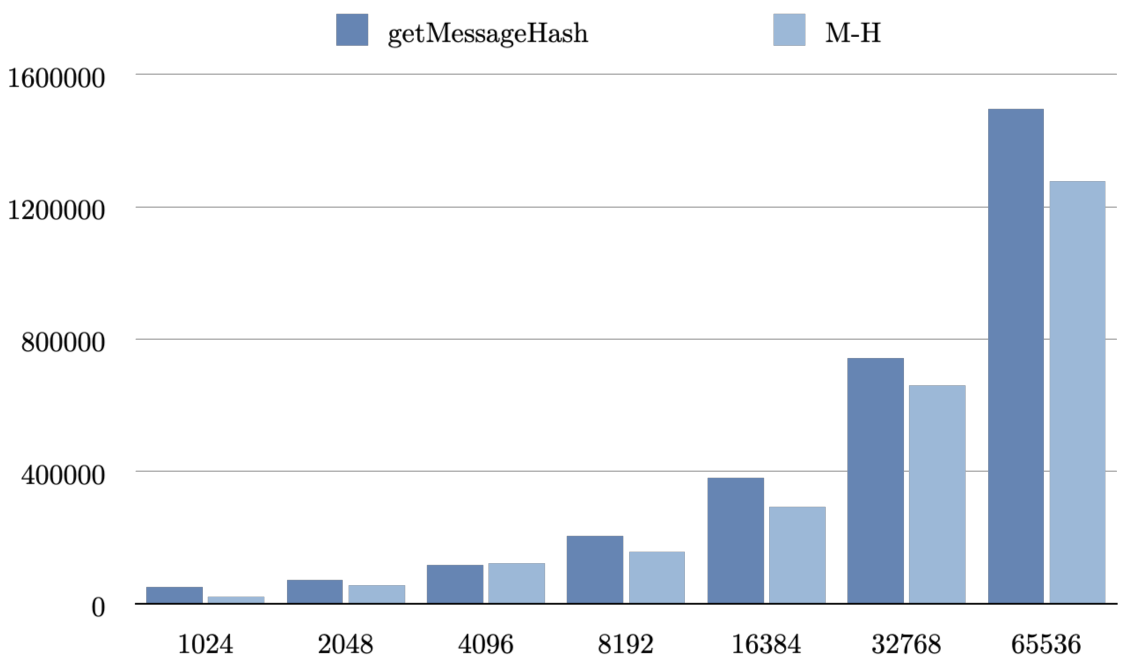


Figure 4.3: The cost difference between *validateSignatureM* and *validateSignatureH* compared with the *getMessageHash* cost

4.5.3 Cost Motivations

Operation	Gas
ADD	3
MUL	5
SUB	3
DIV	5
SDIV	5
MOD	5
SMOD	5
ADDMOD	8
MULMOD	8

Table 4.1: Some machine operations and their gas cost

It is important now to point out why the cost is so high. Ethereum has a price for each machine operation, an extract from the Ethereum Yellow Paper's prices table [57] is shown in Table 4.1.

As it is possible to see these from Table 4.1 operations do not cost a lot of gas but a single line of Solidity code is converted in a large amount of machine operations, moreover *mod* operations are more expensive than the others. In the signature validation, every operation is done in modular arithmetic and the numbers used are all numbers with 256 bits. This easy Solidity function does not receive any parameter and does not return anything, it just adds 1 to 1 and costs 133 when called.

```
contract Add {
    function add() public pure {
        uint x = 1 + 1;
    }
}

```

Thanks to RemixIDE [58] it is possible to see the Solidity code translated into machine instructions.

```
    PUSH1 80 - LINE 4
    PUSH1 40 - LINE 4
    MSTORE - LINE 4
    CALLVALUE - LINE 4
    DUP1 - LINE 4
    ISZERO - LINE 4
    PUSH1 0e - LINE 4
    JUMPI - LINE 4
    INVALID -
    DUP1 -
    REVERT -
    JUMPDEST - LINE 4
    POP - LINE 4
    PUSH1 04 - LINE 4
    CALLDATASIZE - LINE 4
    LT - LINE 4
    PUSH1 26 - LINE 4
    JUMPI - LINE 4
    INVALID - LINE 4
    CALLDATALOAD - LINE 4
    PUSH1 e0 - LINE 4
    SHR - LINE 4
    DUP1 - LINE 4
    PUSH4 3dd9ebdf - LINE 4
    EQ - LINE 4
    PUSH1 2a - LINE 4
    JUMPI - LINE 4
    JUMPDEST -
    INVALID -
    DUP1 -
    REVERT -
    JUMPDEST -
    PUSH1 30 -
    PUSH1 32 -
    JUMP -
    JUMPDEST -
    STOP -

```

As with any other programming language, a few lines of code generate a very big number of machine instructions and this is the main motivation why the cost of *validateSignature* is so expensive.

In addition, also the function's parameters represent a cost because each of them is written in the memory, as any other variable, and this constitutes a cost depending on their dimension; 4 gas for each zero byte in the transaction data and 16 gas for each non-zero byte in the transaction. For example, if the data is $0x00ff$ the cost is 4 for the double 0 and 16 for the double f, in total 20 gas for a variable of 2 bytes. Writing in the memory or in the storage is costly, but this gives room to possible small optimizations, paying attention to every variable.

4.5.4 Cost Optimizations

The first thing to do is to declare all the function's parameters that are just read from *memory* to *calldata* this is convenient because as the documentation says: *Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory* [59]. This implies that calldata is cheaper because it is written in a just read memory while a *memory* location since can be modiefied is more expensive to handle. This modification leads to an optimization of the code because almost all the parameters received inside the contract's functions are not modiefied but this even if important change does not intruduce a significant gas cost reduction.

Another optimization that has a small impact on the gas cost even if logically important to implement is to mark the *validateSignature* and the *getCurve* as *external* instead of *public*. In Solidity there are 4 types of modifiers:

- *private*: a function marked with this modifier can only be accessed by the contract it resides within. No other contract can access it.
- *internal*: a function marked with this modifier may be called privately by the contract or by the contracts that inherit from the current one.
- *external*: a function marked with this modifier can only be called outside the contract by the other contracts, or web3 programs.
- *public*: a function marked with this modifier can be called from the inside and the outside of the contract.

The *public* modifier is equivalent to *external + internal* modifiers and this implies that a function marked as public is more expensive because enables the *external + internal* behaviours but in case just the *external* behaviour is needed it is cheaper and logically more correct to enable only it. An additional optimization is an aware use of the variables, declaring useless variables or variables used just a few times should be avoided if possible, maybe reusing already declared ones. This is because every time the program needs space to allocate a variable this space must be paid. In the Verifier every declared variable is needed for the signature verification except for one.

In this line, for example:

```
    (uint256 Qx, uint256 Qy) = addPoints(uGx, uGy, vPubKx, vPubKy);

```

the Qy is not needed in the following code so it can be omitted in this way:

```
    (uint256 Qx, ) = addPoints(uGx, uGy, vPubKx, vPubKy);

```

to avoid paying the allocation of an unused variable.

The last and the most effective optimization is enabling the Solidity compiler optimizer. It is able to optimise expressions and reorder instructions to achieve reduced gas consumption. It can be configured with a parameter called *runs.* This parameter sets the desired kind of optimization from the compiler; if it is set to a low value e.g. 200 the compiler focuses on optimizing the deployment process while if it is set to a high value it optimizes the code focusing on the execution. When the focus of the Optimizer is on the deployment, deploying the contract will be cheaper but this will lead to a higher cost when calling the contract's functions. This is useful when the contract will be called a few times and the deployer wants to save gas from the deployment. On the other hand, optimizing the execution means paying more gas at the deployment, but having cheaper functions. It is very convenient to have a well-optimized code especially when these functions will be called many times because it leads to high gas savings. The parameter *run* represents the number of function calls that the contract is expected to receive.

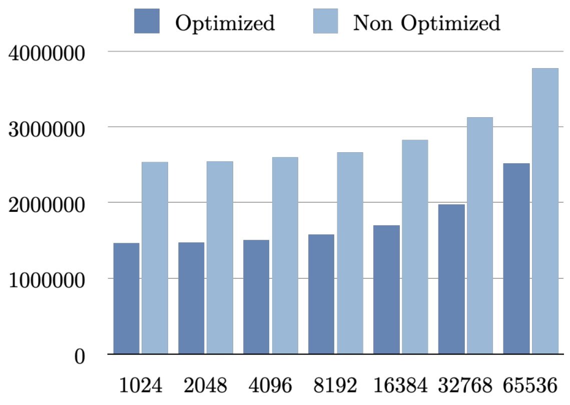


Figure 4.4: Cost Comparisons increasing the message length between optimized with $run = 10000$ and non-optimized *validateSignatureM*

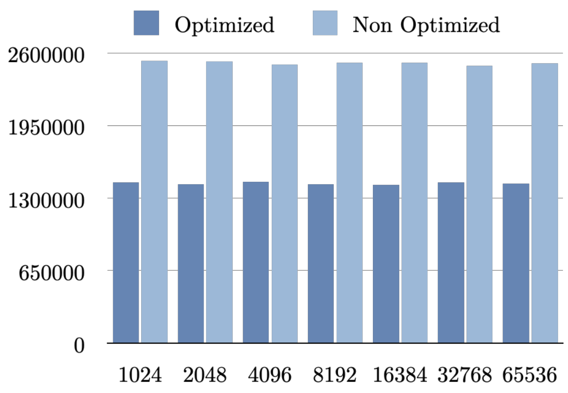


Figure 4.5: Cost Comparisons increasing the message length between optimized with $run = 10000$ and non-optimized *validateSignatureH*

As it is possible to see from Figure 4.4 and 4.5 and the optimized code with the run parameter set to 10000 is a lot cheaper than the non-optimized code. In Figure 4.6 it is shown the gas saving percentage when calling the *validateSignatureM* and the *validateSignatureH* with the run parameter set to 10000 with messages of increasing length. It is interesting to see that the percentage, for the *validateSignatureM*, decreases with the length increment; this is because when incrementing the message length the function requires more gas for computing the hash. So in the total amount of gas used, a bigger part is consumed for computing the hash and the optimization of the function produces a smaller improvement when compared with the case with shorter messages. The smaller improvement is because the hash computation is built-in in Solidity and it cannot be further optimized. When messages are short the gas consumption is mainly due to the signature verification which is also more sensible to the optimization and this explains the higher percentages in gas saving.

On the other contrary, to identify in a more precise way the code improvement is better to look, always in Figure 4.6, at the gas saving percentage of the *validateSignatureH* to understand the optimization of the code excluding the hash computation. As it is possible to see in the graph the *validateSignatureH* remains inside the interval between 44% and 42% and it is possible to consider the gas saving after the optimization of about 43%.

At this point, it is important to measure the effect of varying the value of *runs* on the optimization. After some tests shown in Figure 4.7 it was clear that incrementing too much the value of run will not produce a significant reduction in the gas consumption.

This is due to mainly two motivations. The first and most important is that just enabling the Optimizer with its default and lowest value of 200 allows a huge gas saving, and this means that the Optimizer has already done very good work.

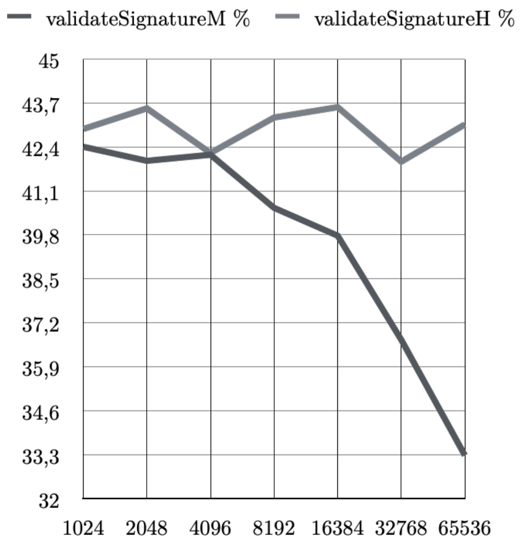


Figure 4.6: Gas saving percentage with $run = 10000$

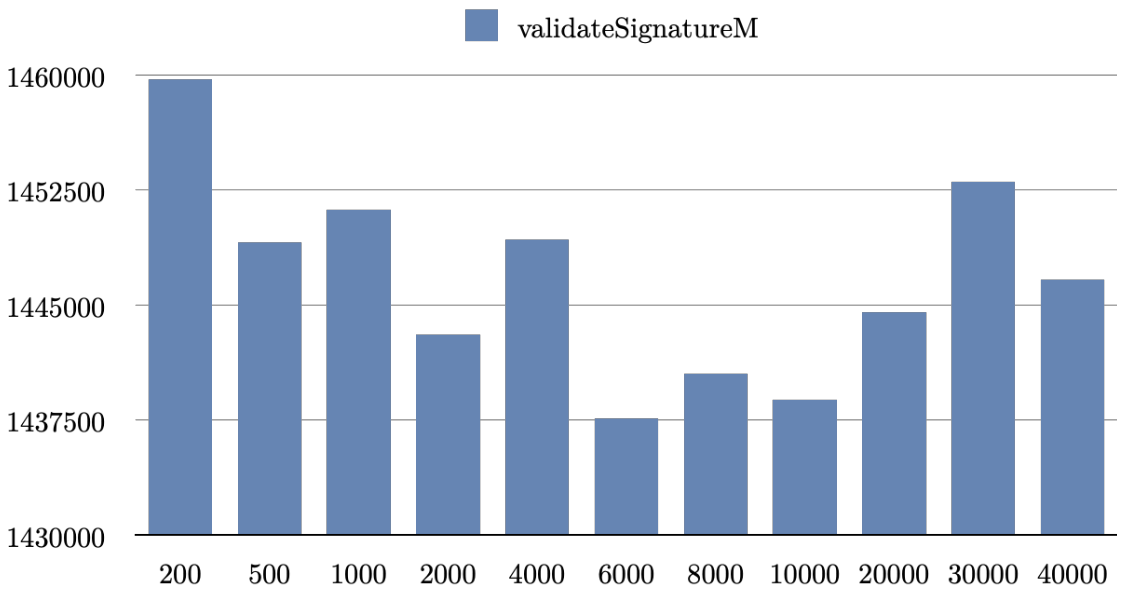


Figure 4.7: Gas cost varying the *run* parameter

The second motivation is that the Optimizer has a threshold beyond which it stops making significant changes. Increasing the runs parameter beyond this threshold may not lead to additional optimizations, instead, it could reduce its effectiveness as seen in Figure 4.7 after the 10000 value.

It is worth noting that all the results in this Section 4.5 are subject to a small variation that is because when calling a function its parameters should be paid, and during the tests, the message, and the keypair are randomly generated so the cost at every call dependents on the values of the parameters following the price as stated in Section 4.5.2. To further explore the impact of this variation another test with 1000 messages was done and it shows that it is quite equal to set the parameter from 6000 to 10000 as represented in Figure 4.8, but the best value is 10000.

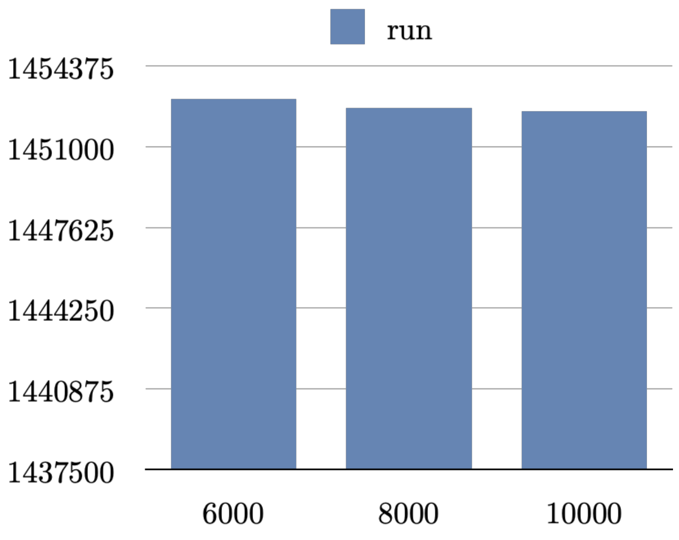


Figure 4.8: Gas cost with 1000 test messages for different *run* values

In conclusion, optimizing code is a crucial aspect of smart contract development. It is necessary to ensure that contracts are cost-effective for the users. The presented approaches can achieve this, but all must be applied for the best results. The success of a contract depends on its optimized code. For the Verifier, all of the optimization techniques have been applied, and the Solidity Optimizer was found to be the most effective. However, it is important to note that this may not be the case for different code.

4.5.5 A Remarkable Comparison

The operations compared with the verifier in section 4.5.1 are common transactions that are executed on a blockchain like Ethereum but none of them implement heavy cryptographical operations. So after having optimized the contract as explained in section 4.5.4 to achieve its cheaper version, it is interesting to compare it with another contract that implements heavy cryptography, a good example is the Polygon On-chain ZK Verification contract [60]. In this tutorial [61] from Polygon, it is possible to simulate the airdrop of an ERC-20 token[6] where only people older than 20 years old can claim it. The Zero Knowledge Proof is a protocol to prove a property or knowledge without the need to disclose it, in the example, it is possible to prove to be older than 20 without exactly telling the current age. This kind of protocol is very helpful to maintain the privacy of the users, it allows to give services to users with particular requirements without the need to disclose more details than needed. Another example could be related to financial services, when there is the need to perform the tax return, in Italy, the amount of taxes to pay depends on the total amount of the income but the are income bands that establish the amount to pay. This scenario is a perfect example of the usefulness of the ZKP because citizens do not have to declare all their assets but just prove to belong to a certain band.

To compare the Verifier's cost with the cost of this contract is it possible to look for the contract address that is available in the tutorial [61] and look for it in Polygonscan[7] [64]. In the tutorial, it is also present the link to Poligonscan of a ZK request transaction [65], while it is possible to find a ZK response [66] by analyzing the Poligonscan page of the contract.

As it is possible to see from Figure 4.9 the cost of a signature validation with a message of 1024 bytes is a bit more expensive than the ZKPresponse, but it is fairly cheaper than the sum of ZKPrequest and ZKPresponse. In particular, the fact that the cost of a validation is comparable to the ZKPresponse indicates that its cost is all in all acceptable, although rather high compared to more common operations.

[6]The ERC-20 introduces a standard for Fungible Tokens [62]

[7]PolygonScan is a blockchain explorer specifically designed for the Polygon network, which is a layer 2 scaling solution for Ethereum. [63]

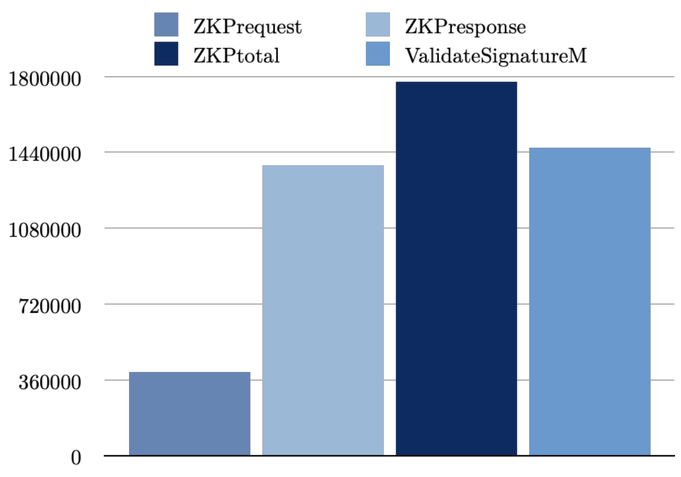


Figure 4.9: Comparison between ZKP contract and validateSignatureM with a Message on 1024 bytes

4.6 Time Evaluation

4.6.1 Time in distributed environment

An important aspect of a verifier is the time required for a response, usually verifying a signature is a fast operation performed on a local machine. First of all, it is important to point out that Proof of Stake blockchains insert blocks with a defined frequency.

Ethereum actually has a block time of approximately 12 seconds. However, as illustrated in Figure 4.10, the block time was not perfectly constant. This was due to Ethereum's initial status as a Proof of Work blockchain, resulting in a probabilistic block time that depended on the miner's computing power and speed (or luck) in solving the Proof of Work puzzle. This puzzle is designed to take almost the same amount of time to solve, regardless of changes in network computation power. However, there may still be situations where a solution cannot be found quickly. In addition, Ethereum must execute smart contract code, which can be time-consuming. For instance, as shown in Figure 4.10, there was a significant increase in block time in 2017. This was due to the exponential growth in Ethereum's popularity, leading to a high volume of users utilizing dApps and saturating the EVM's computational capacity. The significant increase in the use of saturated blocks and computational power led to an increase in gas fees. Users who wanted to prioritize their transactions paid higher fees, but the fees became too high, causing demand to decrease and gas fees to lower. As a result, the community began to consider Ethereum's scalability.

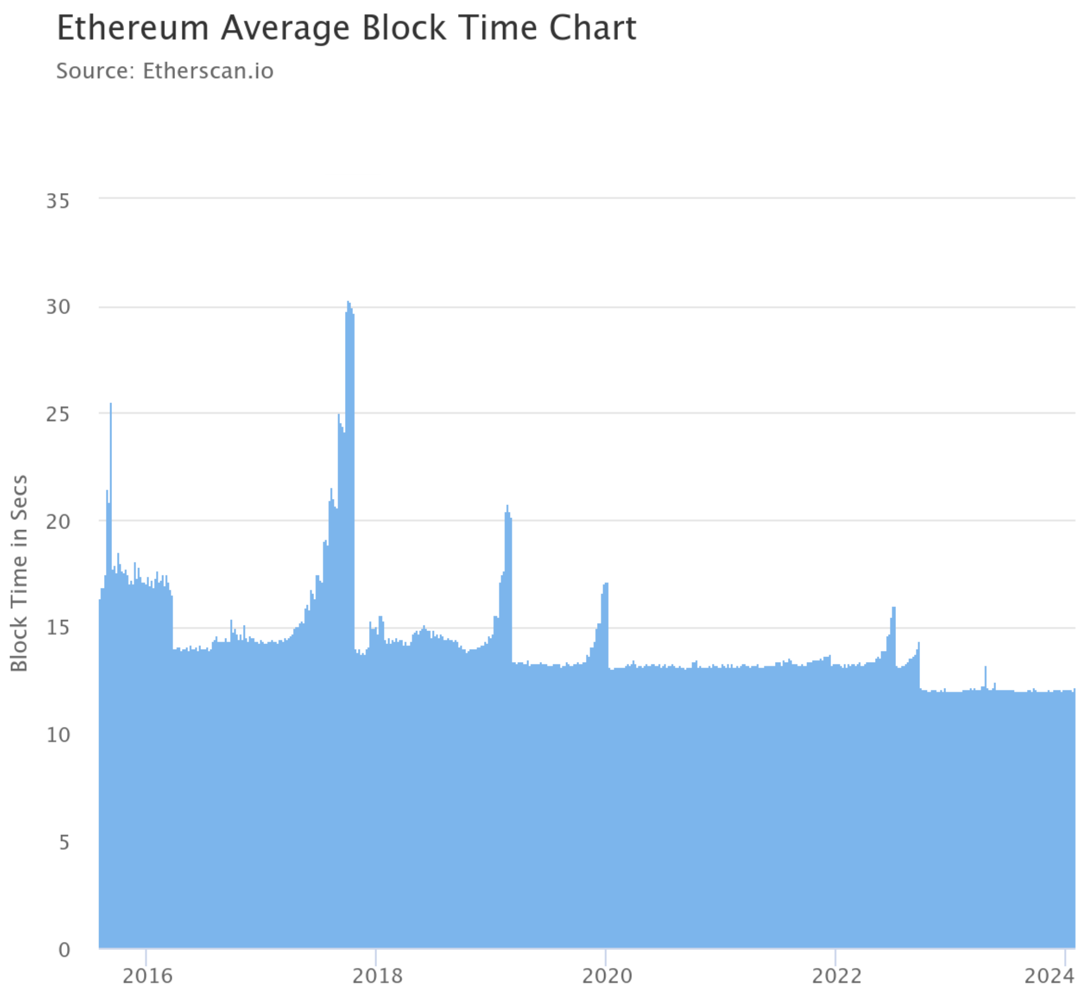


Figure 4.10: Average block time for Ethereum, from its start in July 2015 to January 2024 [67]

These events repeated periodically, albeit with a smaller impact. These discussions resulted in the largest hard fork ever seen in a blockchain. On September 15th, 2022, Ethereum changed its protocol from Proof of Work to Proof of Stake. As shown in Figure 4.10, the block time has remained constant, to 12 seconds, since that date, thanks to the Proof of Stake protocol.

In IOTA ISC Shimmer, the average insertion time is about 4.7 seconds [68], even though the developers say in their documentation that they aim for a sub-second block time [69], it is also important to note that they explicitly say in the documentation that the block time is not enforced:
There is no guaranteed block time. . . . ISC does not enforce an average block time.

This means that block times are variable; a new block will be created as soon as needed [70]. The value of 4.7 seconds is computed over the whole working period of the Shimmer EVM chain but the actual value strongly depends on the network state and for example at the moment of writing this thesis is 2.4 seconds [71].

To better understand the time needed to validate a signature some tests have been performed to estimate the values and compare the obtained ones with the declared in the documentation, these tests took place inside test networks and not inside mainnets because of the cost of the native tokens, anyway, the purpose of test networks is to test code but usually a testing network is slower than a main one so the obtained values can be interpreted as a worst case for a main network.

Between these tests with testnets also the local deployment of IOTA ISC has been used to be compared with Sepolia and ShimmerEVM testnet. The results of these tests have been summarised in Table 4.2. The expected values are taken at the moment of writing, for Sepolia from [72] looking for every block to the *in value*; for ShimmerEVM Testnet from [73] while the expected time for the local deployment is the actual time of the ShimmerEVM mainnet from [71] because this local network is intended to work as the main one; for IOTA environment as said before it is important to remember that there is no a guaranteed block time because the network tries to do its best. It is extremely important to point out that even if the tests show that one signature verification is performed every 12 seconds, two or more transactions from the same user could be inserted in the same block so even if the waiting time is constant at 12 seconds the verifications obtained are more than one. With the current situation of the Ethereum/Sepolia network, it is hard for a user to have more than one transaction in the same block. On the contrary, having a signature verification every block is a good average situation for a user, because the wait could be longer than a block. Moreover, if a lot of users use the Verifier and if a block is completely filled with transactions towards it, the number of transactions, that cost $1.5M$ of Gas, in the block is between 10 and 20 because the dimension of a block is usually $15M$ of Gas but it can arrive up to $30M$ [74], remember that having full blocks increases the base fee as explained in Section 2.3.1. On the other hand, in IOTA ShimmerEVM there is only one transaction per block. [69]

Blockchain	Measured Time (s)	Expected Time (s)
Sepolia	12	12
ShimmerEVM testnet	1,7	19,6
Local ShimmerEVM	1,2	2,4

Table 4.2: Block time obtained from tests and expected

As shown in the table, Sepolia is aligned with the expected time, but it has the longest waiting time overall. When using the ShimmerEVM testnet, the block time may be faster than expected due to low network usage. Therefore, the average value computed for the day may not accurately represent the real block time, unlike in Sepolia. The expected value for the mainnet and testnet is the sum of the block time and the wait for a new block taken as a mean, and if the network is not used this time increases. Additionally, the measured time corresponds exactly to the time elapsed from when the transaction is sent to when the answer is received. The same considerations apply to the local chain, with some clarifications. The local deployment was completely free during testing, so it provided the best available performance. The expected value is that of the mainnet because the purpose of the local deployment was to represent the mainnet, not the testnet. It is important to note that the network can operate faster, but this is not commonly used. This explains why the measured values for the ShimmerEVM Testnet and local deployment are both very close and very far from the expected value. During the tests, transactions were sent quickly one after the other, and the network was able to handle them all very quickly, faster than the average time.

4.6.2 Time in local environment

On the other hand, it is interesting to understand the time required for signature verification when computed on a local machine such as a server or a Raspberry Pi. The times measured in these tests represent just the measurement of the time needed for the signature validation, these are just the execution times.

The tests were performed on:

- a server equipped with an Intel Xeon 2.10GHz processor
- a Raspberry Pi with the addition of a TPM module
- a STM32[8]

The results, summarised in Table 4.3, show that the server is incredibly faster than any other equipment used while the Raspberry is also very fast and ten times faster than ShimmerEVM Testnet, on the contrary, the STM32 is faster than Ethereum and Sepolia[9], otherwise it is the slowest one, anyway this is a very small and low powerful microcontroller.

In conclusion, a server is faster than a distributed ledger, but it is important to carry out signature verification in a secure environment. Protecting a server and its

[8]The STM32 family of 32-bit microcontrollers is based on the Arm Cortex-M® processor by STMicroelectronics. [75]

[9]Take in mind that the block time of 12 seconds is a fixed choice of the community but it is not necessarily the best time obtainable with EVM.

Machine	Measured Time (ms)	Measured Time (s)
Server	1,9	0,0019
Raspberry Pi TPM	178	0,178
STM32	6031	6,031

Table 4.3: Signature Verification time on computers

network is a complex task that may introduce configuration errors or vulnerabilities in the infrastructure. Avoiding all of these issues is possible, but it can be a costly and time-consuming activity for a company. On the other hand, using a blockchain could be a possible solution for the reasons that will be presented in Chapter 5.

Chapter 5

Final Considerations

5.1 Final Considerations on the Verifier

The primary objective of this work was to determine the feasibility of implementing signature verification using a smart contract. The answer is yes, but there is the need to select an appropriate technology. Ethereum is the most popular platform for deploying and testing smart contracts. It supports Solidity, a strongly typed language that natively implements some operations such as the hash and modular arithmetic, but with some limitations that, as discussed in Section 4.3.4, introduce the need to switch from affine to Jacobian coordinates. By using Jacobian coordinates, errors presented in Section 4.3.4 can be avoided, and gas consumption can be optimized using computationally lighter operations.

To improve performance, IOTA ISC can be used instead of Ethereum. IOTA ISC is compatible with the Ethereum technology thanks to the EVM contract, as explained in Section 2.2.8, but it is significantly faster, as shown in Table 4.2. Although not as fast as a server or a Raspberry Pi, the wait time for a signature check is still faster than on an embedded machine like the STM32. Waiting less than 2 seconds is a reasonable amount of time, considering the benefits and ease of sending a request to a smart contract.

5.2 Final Considerations on Possible Utilization

Moving from a centralised server to a blockchain could offer several advantages for a company. For instance, consider a start-up that needs to configure services within a secure and properly configured server. This process has various costs, from purchasing a suitable server to configuring it securely and providing the services the company offers.

Initially, the server may not receive many requests, but as the company gains popularity and the number of customers increases, the server may become inadequate to serve all users. This could result in the need to purchase a new server, which is not only more expensive but also requires reconfiguration from scratch.

However, if the company's services can be coded into a smart contract, the company can benefit from its use. Dealing with a server is not a problem, and the company may choose to deploy a blockchain node, but it is not necessary. The main costs to consider are the development of the smart contract and the gas cost when invoking its functions. The cost of the contract's functions is charged to the caller and can be handled in two ways, depending on who calls the function. If the function is called to provide a service within the company, such as verifying a signature, the Verifier smart contract presented in Chapter 4 could be used. In this case, the company would have to pay for the gas but would save the cost of a server. The company requires a secure and reliable method for verifying signatures, while also aiming to minimise server costs by avoiding complex configuration, maintenance, and high power consumption. To achieve this, the founders could decide to use the Verifier Smart Contract, which is already available on the blockchain, thus eliminating the need for them to implement it themselves. The contract enables users to pay only for the gas consumed during signature verification. The fact that the contract is deployed on the blockchain has additional benefits. The code of the smart contract is public, allowing all blockchain users to verify it. This ensures that any issues with the code can be detected by a large group of people. If the issue has not been resolved the community will probably identify it and discourage the smart contract's usage[1].

In contrast, if the service is designed to be used by external users, the customer would have to pay for the gas and the company. This double cost is likely to discourage users from using the contract, and in addition, the contract is public on the blockchain, so it can be used by anyone without paying the developer. The company could develop a system to address these issues by generating revenue from user contract usage and implementing a tariff that covers both gas and company fees. However, this topic was not covered in this work, which focused solely on the use of smart contracts within the company which deploys and uses the smart contract. To address this issue, the company could consider creating a token that users can purchase to pay for the service when they request it.

[1]Within the blockchain, all data is immutable. Therefore, to update any code after modification, it must be redeployed.

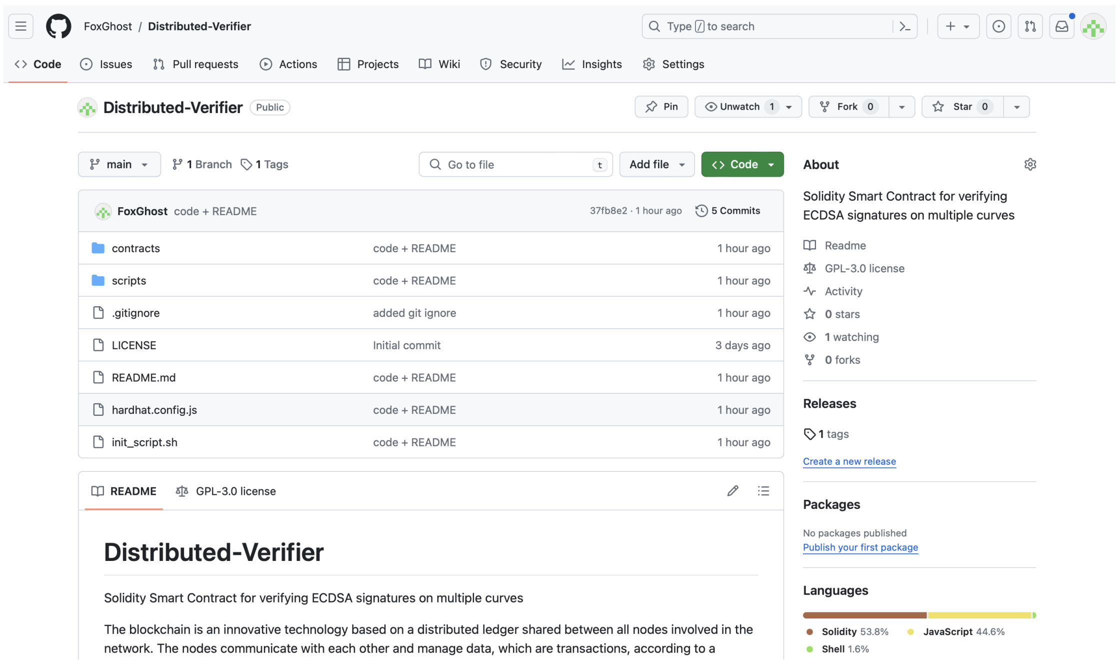
5.3 Future Work

The purpose of this thesis is to develop a Verifier for research purposes. However, for a real application of the Verifier, the contract must already be aware of the public keys that will sign messages. To achieve this, a data structure should be defined to collect all the keys, along with a method to update it. Alternatively, it may be decided to define it only at the deployment stage, without allowing any further modification. Furthermore, the validateSignature function should no longer receive the public key as an input parameter. Instead, it should have a method to retrieve the public key associated with the user who requests the verification. This is a crucial aspect as it allows for the decoupling of the public key used to sign the transaction from the private key used to sign the message.

Appendix A

Code Repository

The developed code can be found in GitHub at:
https://github.com/FoxGhost/Distributed-Verifier
tag: *thesis*



FoxGhost code + README 37fb8e2 · 1 hour ago 5 Commits

contracts	code + README	1 hour ago
scripts	code + README	1 hour ago
.gitignore	added git ignore	1 hour ago
LICENSE	Initial commit	3 days ago
README.md	code + README	1 hour ago
hardhat.config.js	code + README	1 hour ago
init_script.sh	code + README	1 hour ago

About

Solidity Smart Contract for verifying ECDSA signatures on multiple curves

Readme
GPL-3.0 license
Activity
0 stars
1 watching
0 forks

Releases

1 tags
Create a new release

Packages

No packages published
Publish your first package

Languages

Solidity 53.8% JavaScript 44.6% Shell 1.6%

Distributed-Verifier

Solidity Smart Contract for verifying ECDSA signatures on multiple curves

The blockchain is an innovative technology based on a distributed ledger shared between all nodes involved in the network. The nodes communicate with each other and manage data, which are transactions, according to a

Appendix B

Hornet Deployment

This appendix presents the steps to run a Private Tangle in the local environment. According to the documentation, there are only a few steps to be taken. [76].

B.1 Starting

Initially download from the official IOTA repository the Hornet code.

```
    git clone https://github.com/iotaledger/hornet.git

```

Now move to the */hornet/private_tangle/* folder which contains a Docker-based setup to run a Private Tangle. Then run these commands:

```
    ./bootstrap.sh
    ./run.sh

```

The *./bootstrap.sh* script bootstraps a Private Tangle by creating the Genesis snapshot and required files. The *./run.sh* script activates 2 nodes and a Coordinator, optionally it can receive as parameter 3 or 4 to spin up 3 or 4 Hornet.

Appendix C

Wasp Deployment

C.1 Starting

Initially download from the official IOTA repository the Wasp code.

```
    git clone https://github.com/iotaledger/wasp.git

```

Now move to the */wasp/tools/local-setup* folder which contains a Docker-based setup to run a Wasp local setup. The folder contains a *docker-compose.yml* file which spins up a private Hornet node, together with a container running the Wasp node and another one running the Wasp-dashboard. Everything is regulated by a Traefik reverse proxy.

C.2 Attach the WASP to existing private tangle

Since the Hornet deployment was been already done in the working environment, it is necessary to remove everything related to the Hornet node from the *docker-compose.yml.* In particular, remove the hornet-nest service. It is extremely important that containers that are going to be deployed must be placed on the docker network of the private Tangle (the Hornet nodes already present) otherwise the Wasp container will not be able to attach to the Hornet's inx address.

The Wasp's *docker-compose.yml* must have a network section like this:

```
version: "3.9"
networks:
    tangle:
        name: private_tangle_peering_net
        external: true

```

To know the network name run in the terminal *docker network ls* while the hornet nodes are running. It returns all the docker networks the one that looks like the name of the parent folder of the Hornet's *docker-compose.yml* is the needed one.

The Wasp's *docker-compose.yml* should now contain the *network section* and the *services section* which includes *Traefik, Wasp and Wasp Dashboard*

C.2.1 Addresses and Ports

The standard deployment runs with these ports:

- Wasp:
  - API: http://localhost:9090
- Hornet:
  - API: http://localhost:14265
  - Dashboard http://localhost:80

When deploying on a machine in a local network with an ipv4 private address there is the need to change some network parameters:

1. replace every occurrence of localhost with the private ipv4 address where the node will be deployed, the Wasp Dashboard will be available in the local network at the address: http//192.168.*.*/wasp/dashboard
2. the Hornet's inx address has to be correctly set in the Wasp's *docker-compose.yml*, that must be known based on the one declared on the Hornet's *docker-compose.yml*. For example in the Wasp's *docker-compose.yml*:

```
--inx.address=172.18.211.13:9029

```

3. if port 80 is already used in the environment do the following modifications:
   - in the Traefik component:

```
#from
ports:
    - "${HTTP_PORT:-80}:80/tcp"

#to
ports:
    - "${HTTP_PORT:-90}:80/tcp"

```

   - in the Wasp Dashboard:

```
#from
environment:
        - WASP_API_URL=http://192.168.94.14/wasp/api
        - L1_API_URL=http://192.168.94.14
#to
environment:
        - WASP_API_URL=http://192.168.94.14:90/wasp/api
        - L1_API_URL=http://192.168.94.14:90

```

C.3 Running

After doing the preceding modifications run:

```
docker-compose pull

```

to fetch the dependencies, then create the needed volumes:

```
docker volume create --name wasp-db

```

To start the setup run:

```
docker-compose up -d

```

After startup, the Wasp Dashboard is visible at: *http://localhost/wasp/dashboard/*

C.4 Stopping

To stop the environment run:

```
docker compose down

```

to also remove the volumes:

```
docker volume rm wasp-db hornet-nest-db

```

note that this will remove all the databases thus all data will be lost.

C.5 Naming

Giving a specific name to every Wasp node could be useful, to do this remember to modify the following parameter with the desired name, for example: *container_name: wasp-1.* To refer to a node inside the docker network with a name instead of an address remember to add this string under the field container name *hostname: wasp-1.*

C.6 Multi Machine Deploy

To have a local blockchain, having many nodes is desirable so deploying nodes on different machines can be easily done. Every machine is supposed to already have a Horner running node, and the just described deploy method can be easily done on every machine just paying attention to ipv4 address and port configuration inside Wasp's *docker-compose.yml.*

Appendix D

Cost

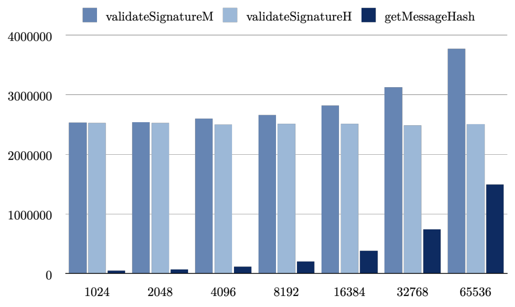


Figure D.1: Shimmer

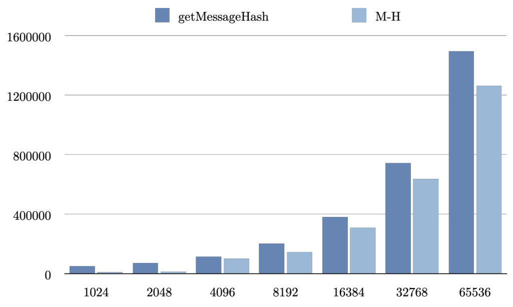


Figure D.2: Shimmer

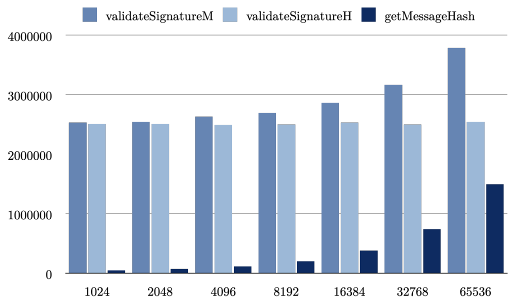


Figure D.3: Sepolia

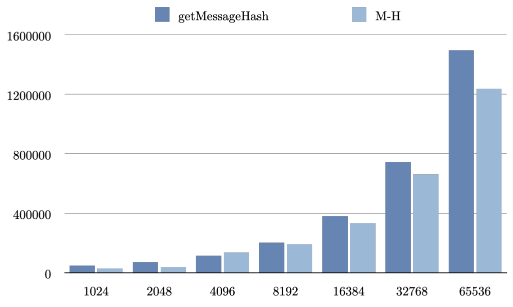


Figure D.4: Sepolia

Figure D.5: On the left the validateSignatureM cost grows with the message length. On the right the cost increment compared with the getMessageHash cost. Every row shows a different test chain.

Bibliography

[1] Carlo Marroni. *Tremonti: Ipo di Coinbase? «Hanno quotato il nulla, certificato dal nulla»*. Apr. 2021. URL: `https://24plus.ilsole24ore.com/art/tremonti-l-ipo-coinbase-hanno-quotato-nulla-certificato-nulla-AE3jo1`. (accessed: 19.3.2024) (cit. on p. 1).

[2] Mattia Feltri. *Oche e monete*. Apr. 2021. URL: `https://www.lastampa.it/topnews/firme/buongiorno/2021/04/17/news/oche-e-monete-1.40160858/`. (accessed: 19.3.2024) (cit. on p. 1).

[3] Michele Mandelli. *La marcia d'avvicinamento tra le banche e il mondo delle criptovalute*. Oct. 2022. URL: `https://www.ilsole24ore.com/art/la-marcia-d-avvicinamento-banche-e-mondo-criptovalute-AEBzC6CC?refresh_ce=1`. (accessed: 19.3.2024) (cit. on p. 1).

[4] Lorenzo Magnani. *Banche e blockchain: la finanza italiana è sempre più crypto*. Apr. 2022. URL: `https://www.we-wealth.com/news/aziende-e-protagonisti/aziende-e-protagonisti/banche-e-blockchain-la-finanza-italiana-e-sempre-piu-crypto`. (accessed: 19.3.2024) (cit. on p. 1).

[5] Marshall Langer, Alessandro Villadei, and Valerio Mancini. *Gli investimenti nella tecnologia blockchain arriveranno a 11.7 miliardi di dollari quest'anno*. Apr. 2022. URL: `https://romebusinessschool.com/it/blog/gli-investimenti-nella-tecnologia-blockchain-arriveranno-a-11-7-miliardi-di-dollari-questanno/`. (accessed: 19.3.2024) (cit. on p. 1).

[6] Bitcoin Community. *Proof of work*. URL: `https://en.bitcoin.it/wiki/Proof_of_work`. (accessed: 12.2.2023) (cit. on p. 3).

[7] Bitcoin Community. *The Byzantine Generals Problem*. URL: `https://en.bitcoin.it/wiki/The_Byzantine_Generals_Problem`. (accessed: 12.2.2023) (cit. on p. 4).

[8] Leslie Lamport, Robert Shostak, and Marshall Pease. «The Byzantine Generals Problem». In: ACM Transactions on Programming Languages and Systems, vol. 4, n. 3 (July 1982), pp. 382–401 (cit. on p. 4).

[9] Ethereum Community. *Proof Of Stake (POS)*. URL: https://ethereum.org/developers/docs/consensus-mechanisms/pos. (accessed: 12.2.2023) (cit. on p. 5).

[10] IOTA Foundation. *The Coordinator - PoA Consensus*. URL: https://wiki.iota.org/learn/protocols/coordinator/. (accessed: 14.2.2023) (cit. on p. 6).

[11] IOTA Foundation. *White Flag Ordering*. URL: https://wiki.iota.org/tips/tips/TIP-0002/. (accessed: 14.2.2023) (cit. on p. 7).

[12] KPMG. *Confronting Complexity: How Business Globally is Taking on the Challenges and Opportunities*. Jan. 2011. URL: https://erm.ncsu.edu/library/article/complexity-challenges-risk/. (accessed: 19.3.2024) (cit. on p. 8).

[13] Blockchain.com. *Hashrate Distribution*. URL: https://www.blockchain.com/explorer/charts/pools. (accessed: 19.3.2024) (cit. on pp. 9, 10).

[14] Ethereum Community. *IS PROOF-OF-STAKE SECURE?* URL: https://ethereum.org/developers/docs/consensus-mechanisms/pos/faqs#is-pos-secure. (accessed: 19.3.2024) (cit. on p. 9).

[15] IOTA Foundation. *The Transparency Compendium*. URL: https://blog.iota.org/the-transparency-compendium-26aa5bb8e260/. (accessed: 14.2.2023) (cit. on p. 10).

[16] IOTA Foundation. *Identities and Sybil protection*. URL: https://blog.iota.org/identities-and-sybil-protection-in-iota-9c62916ff374/. (accessed: 14.2.2023) (cit. on p. 10).

[17] Nick Szabo. *Smart Contracts*. 1994. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html. (accessed: 28.11.2023) (cit. on p. 13).

[18] Nick Szabo. *Smart Contracts: Building Blocks for Digital Markets*. 1996. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html. (accessed: 28.11.2023) (cit. on p. 13).

[19] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum*. URL: https://github.com/ethereumbook/ethereumbook/blob/develop/07smart-contracts-solidity.asciidoc#what-is-a-smart-contract. (accessed: 19.3.2024) (cit. on p. 15).

[20] Ethereum Community. *ETHEREUM VIRTUAL MACHINE (EVM)*. Sept. 2023. URL: https://ethereum.org/en/developers/docs/evm/. (accessed: 19.3.2024) (cit. on pp. 16, 17).

[21] Ethereum Community. *OPCODES FOR THE EVM*. June 2023. URL: `https://ethereum.org/en/developers/docs/evm/opcodes/`. (accessed: 19.3.2024) (cit. on p. 17).

[22] IOTA Foundation. *The blob Contract*. URL: `https://wiki.iota.org/wasp-wasm/reference/core-contracts/blob/`. (accessed: 14.2.2023) (cit. on pp. 29, 30).

[23] IOTA Foundation. *Core Contracts*. URL: `https://wiki.iota.org/wasp-wasm/reference/core-contracts/overview/`. (accessed: 14.2.2023) (cit. on p. 30).

[24] IOTA Foundation. *The root Contract*. URL: `https://wiki.iota.org/wasp-wasm/reference/core-contracts/root/`. (accessed: 14.2.2023) (cit. on p. 30).

[25] IOTA Foundation. *The accounts Contract*. URL: `https://wiki.iota.org/wasp-wasm/reference/core-contracts/accounts/`. (accessed: 14.2.2023) (cit. on p. 30).

[26] IOTA Foundation. *The blocklog Contract*. URL: `https://wiki.iota.org/wasp-wasm/reference/core-contracts/blocklog/`. (accessed: 14.2.2023) (cit. on p. 30).

[27] IOTA Foundation. *The governance Contract*. URL: `https://wiki.iota.org/wasp-wasm/reference/core-contracts/governance/`. (accessed: 14.2.2023) (cit. on p. 30).

[28] IOTA Foundation. *The errors Contract*. URL: `https://wiki.iota.org/wasp-wasm/reference/core-contracts/errors/`. (accessed: 14.2.2023) (cit. on p. 30).

[29] IOTA Foundation. *The evm Contract*. URL: `https://wiki.iota.org/wasp-wasm/reference/core-contracts/evm/`. (accessed: 14.2.2023) (cit. on p. 30).

[30] IOTA Foundation. *EVM/Solidity Based Smart Contracts*. URL: `https://wiki.iota.org/wasp-evm/introduction/`. (accessed: 14.2.2023) (cit. on p. 30).

[31] Ethereum Community. *GAS AND FEES*. Aug. 2023. URL: `https://ethereum.org/en/developers/docs/gas/`. (accessed: 19.3.2024) (cit. on p. 31).

[32] William Stallings. *Cryptography and Network Security*. Seventh Edition. Pearson, 2017 (cit. on pp. 35, 36, 38, 41, 42, 45).

[33] Taher Elgamal. «A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms». In: IEEE Transactions on Information Theory No. 31 (July 1985), pp. 469–472 (cit. on p. 39).

[34] Claus Schnorr. «Efficient Signature Generation by Smart Cards». In: Journal of Cryptology, No. 3 (Jan. 1991), pp. 161–174 (cit. on p. 39).

[35] Peter Shor. «Algorithms for quantum computation: discrete logarithms and factoring». In: Proceedings 35th Annual Symposium on Foundations of Computer Science (Nov. 1994), pp. 124–134 (cit. on p. 39).

[36] NIST. *Digital Signature Standard (DSS) V4*. URL: `https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf`. (accessed: 19.3.2024) (cit. on p. 39).

[37] NIST. *Digital Signature Standard (DSS) V3*. URL: `https://csrc.nist.gov/files/pubs/fips/186-3/final/docs/fips_186-3.pdf`. (accessed: 19.3.2024) (cit. on p. 43).

[38] Solidity Team. *Solidity Programming Language*. URL: `https://soliditylang.org`. (accessed: 19.3.2024) (cit. on p. 49).

[39] Hardhat Organization. *Hardhat Documentation*. URL: `https://hardhat.org/docs`. (accessed: 19.3.2024) (cit. on p. 49).

[40] Solidity Team. *Solidity Documentation*. URL: `https://docs.soliditylang.org/en/v0.8.7/index.html#`. (accessed: 19.3.2024) (cit. on p. 50).

[41] Centre for Research on Cryptography and Security. *Standard curve database - secp256k1*. URL: `https://neuromancer.sk/std/secg/secp256k1`. (accessed: 19.3.2024) (cit. on p. 51).

[42] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum*. URL: `https://github.com/ethereumbook/ethereumbook/blob/develop/04keys-addresses.asciidoc#elliptic-curve-cryptography-explained`. (accessed: 19.3.2024) (cit. on p. 51).

[43] Bitcoin Community. *Secp256k1*. URL: `https://en.bitcoin.it/wiki/Secp256k1`. (accessed: 19.3.2024) (cit. on p. 51).

[44] Centre for Research on Cryptography and Security. *Standard curve database - P-256*. URL: `https://neuromancer.sk/std/nist/P-256`. (accessed: 19.3.2024) (cit. on p. 51).

[45] Trusted Computing Group. *TCG Algorithm Registry*. URL: `https://trustedcomputinggroup.org/wp-content/uploads/TCG-Algorithm-Registry_R1.33_Pub.pdf`. (accessed: 19.3.2024) (cit. on p. 51).

[46] codezone. *Code Duplication: Causes, Consequences, and Solutions*. Nov. 2023. URL: `https://blog.stackademic.com/code-duplication-causes-consequences-and-solutions-1f6fe67ac3d5`. (accessed: 10.2.2023) (cit. on p. 53).

[47] Team Codegrip.Tech. *How Code Duplication Impacts Software Maintainability.* URL: `https://www.codegrip.tech/productivity/how-code-duplication-impacts-software-maintainability/`. (accessed: 10.2.2023) (cit. on p. 53).

[48] Firo Organization. *Big Number Library for Solidity.* URL: `https://github.com/firoorg/solidity-BigNumber/tree/master`. (accessed: 19.3.2024) (cit. on p. 54).

[49] DoubleOdd Group. *Affine and Jacobian (x, w) Coordinates.* URL: `https://doubleodd.group/formulas-xw.html`. (accessed: 19.3.2024) (cit. on p. 55).

[50] Witnet Organization. *Elliptic Curve arithmetic operations written in Solidity.* URL: `https://github.com/witnet/elliptic-curve-solidity`. (accessed: 19.3.2024) (cit. on p. 56).

[51] NIST. *Cryptographic Algorithm Validation Program.* URL: `https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/component-testing#ECDSASigGenPrim`. (accessed: 19.3.2024) (cit. on p. 57).

[52] NIST. *ECDSA Signature Generation Component Testing.* URL: `https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/components/186-3ecdsasiggencomponenttestvectors.zip`. (accessed: 19.3.2024) (cit. on p. 57).

[53] NIST. *Frequently Asked Questions For the Cryptographic Algorithm Validation Program.* URL: `https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/CAVPFAQ.pdf`. (accessed: 19.3.2024) (cit. on p. 57).

[54] Web3.js Organization. *Web3.js Documentation.* URL: `https://github.com/web3/web3.js/blob/0.20.7/DOCUMENTATION.md#web3ethgettransactionreceipt`. (accessed: 19.3.2024) (cit. on p. 59).

[55] ERC-721 community. *WHAT IS ERC-721?* URL: `http://erc721.org`. (accessed: 19.3.2024) (cit. on p. 62).

[56] Solidity Team. *Recovering the Message Signer in Solidity.* URL: `https://docs.soliditylang.org/en/v0.8.0/solidity-by-example.html?highlight=ecrecover#recovering-the-message-signer-in-solidity`. (accessed: 19.3.2024) (cit. on p. 63).

[57] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* URL: `https://ethereum.github.io/yellowpaper/paper.pdf`. (accessed: 19.3.2024) (cit. on p. 65).

[58] *Remix Ethereum IDE.* URL: `https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.22+commit.4fc1097e.js`. (accessed: 19.3.2024) (cit. on p. 67).

[59] Solidity Team. *Solidity Documentation.* URL: `https://solidity.readthedocs.io/en/latest/types.html?ref=hackernoon.com#data-location`. (accessed: 19.3.2024) (cit. on p. 68).

[60] Polygon. *On-chain verification.* URL: `https://devs.polygonid.com/docs/verifier/on-chain-verification/overview/`. (accessed: 19.3.2024) (cit. on p. 73).

[61] Polygon. *Implement an ERC20 zk airdrop in 20 minutes with Polygon ID.* URL: `https://github.com/0xPolygonID/tutorial-examples/tree/main/on-chain-verification`. (accessed: 19.3.2024) (cit. on p. 73).

[62] Ethereum Community. *ERC-20 Token standard.* URL: `https://ethereum.org/en/developers/docs/standards/tokens/erc-20`. (accessed: 19.3.2024) (cit. on p. 73).

[63] Polygon. *Polygonscan.* URL: `https://mumbai.polygonscan.com`. (accessed: 19.3.2024) (cit. on p. 73).

[64] Polygon. *ZK Contract.* URL: `https://mumbai.polygonscan.com/address/0xecf178144ccec09417412d66e2ecc8a2841ee228`. (accessed: 19.3.2024) (cit. on p. 73).

[65] Polygon. *Transaction Details ZK Request.* URL: `https://mumbai.polygonscan.com/tx/0x2ddb2db7b3d35cf7cdf658209b257fd2a51c49df2249bf46ede8979eb8410ffb`. (accessed: 19.3.2024) (cit. on p. 73).

[66] Polygon. *Transaction Details ZK Response.* URL: `https://mumbai.polygonscan.com/tx/0xe096def0cf64a5e43db120eb743ac4a5147376e3039407b4e316a457fce42f23`. (accessed: 19.3.2024) (cit. on p. 73).

[67] Etherscan.io. *Ethereum Average Block Time Chart.* URL: `https://etherscan.io/chart/blocktime`. (accessed: 19.3.2024) (cit. on p. 75).

[68] IOTA Foundation. *Shimmer EVM Mainnet stats.* URL: `https://explorer.evm.shimmer.network/stats`. (accessed: 14.2.2023) (cit. on p. 75).

[69] IOTA Foundation. *ShimmerEVM.* URL: `https://shimmer.network/evm`. (accessed: 14.2.2023) (cit. on pp. 75, 76).

[70] IOTA Foundation. *No Enforced Block Time.* URL: `https://wiki.iota.org/wasp-evm/getting-started/compatibility/#no-enforced-block-time`. (accessed: 14.2.2023) (cit. on p. 76).

[71] IOTA Foundation. *Shimmer EVM Mainnet explorer.* URL: `https://explorer.evm.shimmer.network`. (accessed: 14.2.2023) (cit. on p. 76).

[72] Etherscan.io. *Sepolia Testnet Explorer*. URL: https://sepolia.etherscan.io. (accessed: 19.3.2024) (cit. on p. 76).

[73] IOTA Foundation. *ShimmerEVM Testnet explorer*. URL: https://explorer.evm.testnet.shimmer.network. (accessed: 14.2.2023) (cit. on p. 76).

[74] Ethereum Community. *BLOCKS - BLOCK SIZE*. URL: https://ethereum.org/en/developers/docs/blocks/#block-size. (accessed: 23.3.2024) (cit. on p. 76).

[75] STMicroelectronics. *STM32 32-bit Arm Cortex MCUs*. URL: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html. (accessed: 19.3.2024) (cit. on p. 77).

[76] IOTA Foundation. *Private Tangle*. URL: https://github.com/iotaledger/hornet/tree/develop/private_tangle. (accessed: 14.2.2023) (cit. on p. 85).