Threshold Cryptography with smart contract using zk-STARKs

Rahul Raghavan Tharammal [a, *], Prashant Nitnaware [b]

[a] *Department of Information Technology, Pillai College of Engineering, Maharashtra, India*
E-mail: trahul22me@student.mes.ac.in ; ORCID: https://orcid.org/0009-0002-7262-8975
[b] *Department of Information Technology, Pillai College of Engineering, Maharashtra, India*
E-mail: pnitnaware@mes.ac.in

Abstract. Threshold Cryptography (TC) is a security technique that divides a secret into multiple pieces, with the original secret only being recoverable when a certain minimum number of these pieces are combined together. This approach boosts security by preventing a single point of failure. In blockchain technology, applying TC enhances security by spreading out cryptographic key management responsibilities, thus reducing vulnerability to attacks. To further strengthen blockchain systems, we propose Threshold cryptography using smart contract. Smart contract can ensure accurate execution of the protocol and allow participation from random users, incentivizing honest behavior. The TC protocol integrates generateProof and reconstructSecret protocol functions based on zk-STARKs, a technology that cut down transaction costs and remove the processing tasks off-chain. With this setup, smart contracts only verify computation accuracy, lightening the load on the blockchain network.

Keywords: marketplace, blockchain technology, Gas Cost, Decentralize application, smart contract.

1. Introduction

Threshold cryptography allows multiple users to work together on cryptographic tasks without needing a central authority [1]. Unlike traditional systems that depend on a trusted third party, threshold cryptography shares trust among all users. In this setup, a group of users can jointly create a private key that they all own a part of, without any one user knowing the complete key. Over the last twenty years, research on Threshold cryptography protocols has attracted considerable interest, as explained in Section 3B. The growing fascination with blockchain technology has led to new possibilities and progress in this field.

Blockchain technology isn't just about cryptocurrencies like Bitcoin [2]; it's used in many different areas [3,4]. More advanced versions, like Ethereum, can run smart contracts – predetermined programs stored in the blockchain. Because they're decentralized, blockchains work well for tasks like reaching agreements, generating random outcomes, and creating secure signatures using complex cryptographic methods involving multiple parties

*Corresponding author. E-mail: trahul22me@student.mes.ac.in

TC protocols are crucial in blockchain technology, especially in smart contracts, which can improve these protocols. Typically, users who misbehave in such protocols are just removed without any consequences. This lack of accountability could encourage attacks on the protocol. However, our paper investigates how smart contracts can improve the quality of the TC protocol by increasing user engagement through the use of crypto-economic incentives. This encourages users to be honest when executing the protocol.

Moreover, it's important for outside parties to confirm that a specific public key meets the requirements of the TC protocol and is linked to a specific group of contributors who jointly created the key. While those involved in the system can easily check this, it presents a challenge for those not familiar with the protocol. As a result, we're looking into using contract to guarantee the proper application of the threshold cryptography and to offer a clear overview of the actions performed by each user.

Using smart contracts can sometimes create extra work, but there are ways to make it easier. One method is called Secure Multiparty Computation with zkProofs (ZKPs). Instead of performing all the calculations directly on blockchain [6], the smart contract can be tested with minimal costs, thereby increasing the speed of transactions. Another helpful technique is threshold cryptography. This spreads out the cryptographic work among several parties, which means each smart contract doesn't have to do as much on its own. This makes it easier to handle complicated tasks while still keeping everything secure and private.

Therefore, we propose a new cryptographic protocol TC using smart contracts. These contracts help users to carry out various stages of the protocol smoothly by managing the process. Instead of direct communication, the blockchain acts as the medium for exchanging information, removing the need for users to connect directly. Additionally, the smart contract includes a simple way to handle disputes by using Secure Multiparty Computation (SMPC) with zk-STARKs. This allows for penalties, like reducing collateral, in case of wrongdoing, ensuring accountability among users. We define the details of the protocol and build a basic version for a blockchain-based EVM. We also investigate the costs associated with implementing the contract, the efficiency and memory usage for proof creations.

The paper is structured like this: Chapter 2 related work, while Chapter 3 gives background information and explains the basic ideas. In Chapter 4 we introduce the protocol, followed by a detailed description of its implementation in Chapter 5. Chapter 6 presents our assessment of the protocol. In end the Chapter 7, we conclude the paper

2. Related Work

Investigation on Threshold cryptography protocols has gained considerable attention in establishing secure threshold cryptosystems. While blockchain technology has begun to integrate threshold cryptography, there is a lack of studies exploring the potential improvements that smart contracts can provide to these protocols.

In 2001, Boneh & Franklin [5] proposed a solution for solving the main escrow problem of an identity based crypto system. Instead of relying on a single entity to issue a user's secret key, they suggested using multiple entities called PKGs. Each PKG gives the user a section of the secret key, and the user combines these parts to obtain a complete secret key. This setup helps avoid key escrow issues because even if some PKGs are dishonest, the user can still reconstruct the key as long as at least one PKG is honest. Boneh & Franklin also mentioned that their method could be expanded to support threshold key issuing using Shamir secret sharing. However, a downside of their method is that all PKGs are seen as equally important. This means that a user has to register with every PKG, which can be difficult in practice. Additionally, the protocol requires secure channels for issuing the partial private keys, which adds complexity and potential security risks.

In 2002, Chen and colleagues [7] as well as Paterson [8] put forward ideas similar to Boneh and his team's approach. However, a major problem with these methods is that every trusted user needs to check and confirm user identities separately, which is just not doable in practice.

In the Cramer-Shoup cryptosystem [9], devised by Cramer and Shoup in 1998, recipients can check the authenticity of a message before decrypting it by using a part of their secret key. This feature makes it easier to share the system securely. Canetti and Goldwasser [10] later refined this system by merging the verification and decryption processes into one step. In their approach, servers use a random process to decrypt messages, computing a value represented as m, $(x'/x)^s$, where s, is a randomly generated value shared among servers (section of the private key), $x$is t the proof in the message, and x' is the proof computed by the servers.

The central unit, decrypting confirms that $(x = x')^s$, In the distributed setup, if the proof is valid, $(x'/x)^s = 1$,, and decryption yields the original message m; otherwise, it produces a random output. However, because the original message m, has no redundancy, it's hard to be certain that the decryption is correct. One way to address this uncertainty is that to decrypt the similar message twice; once results match, it indicates the accuracy of the message. But there's a downside: servers must keep a shared random value s as part of the secret key. This means the key's length increases with each decrypted message. Despite this drawback, the two-step method offers storage advantages and eliminates requirement for a protocol to calculate a mutual value, compared to the alternative method.

The C. Shoup crypto system, developed by Shoup and Gennaro [11], builds on the earlier El Gamal [13] cryptosystem by Lee and Lim [12]. It operates within the random oracle model [14] and aims to prove knowledge of discrete logarithms without interaction, using the Schnorr signature scheme [15]. However, a major challenge emerged: simulating decryption without the secret key required exponential time due to the vast number of potential outcomes, explained by the forking lemma [16]. This obstacle led to a search for stronger assumptions [17] to overcome it. The focus on using unresponsive ZKP of member [18] tends to avoid the complexity of decryption simulation. Since decryption simulation cannot be rewound, techniques similar to those in resettable zero-knowledge proofs were adopted. Rackoff and Simon's proof of knowledge essentially acts as a proof of membership. In this cryptographic setup, similar to other techniques, two keys are used [19]: one for the receiver and one for the sender. With either key, the prover can decrypt and reveal the plaintext. This allows the sender to decrypt the message and allows the proof to be copied without having to go back to the system because it relies on the proof of registration.

While our method may require more resources such as gas costs, processing speed, and memory usage, it proves to be more economically efficient compared to current blockchain solutions. We achieve this by moving complex tasks away from the main blockchain network. Additionally, our approach allows for the integration of additional preprogrammed contracts for advanced cryptographic operations beyond proof verification. This flexibility comes from freeing smart contracts from handling these complicated calculations directly on the blockchain. This simplified method not only improves performance but also optimizes resource distribution, resulting in a more cost-effective solution over time.

3. Background

In this Chapter, we will explain the main idea of the protocol we're suggesting with details explanation of Secret Sharing, and Zero Knowledge Proofs.

3.1. Shamir secret sharing Method

Shamir introduced secret sharing [20] as a method to split a secret s into several pieces, requiring a certain number of pieces for its recovery. It's impossible to reconstruct s from one piece less than the required amount, and the process reveals no information about s. Initially, a dealer chooses a polynomial $f(x) = p_0 + p_1 x + \cdots + p_{k-1^{x^{k-1}}}$, where $p_0 = s$, and randomly [21] selects the subsequent coefficients p_1 through p_{k-1}. Then, the dealer calculates $f(p_i)$ for $p_i = 1$ across a finite field, sending the resulting points $(p_i, f(p_i))$ to users through a secure channel. Polynomial interpolation ensures that for k distinct points, there's a unique polynomial passing through them. By combining the efforts of k shareholders, they can reconstruct the polynomial and find $f(0)$, thus recovering s. However, Shamir's scheme requires trust in the dealer, who might distribute inaccurate shares, and shareholders could present invalid shares during reconstruction. Verifiable Secret Sharing (VSS) schemes address this challenge, with its origins traced back to the work of Chor et al [22]. Various VSS schemes, like those introduced by different researchers [23,24,25,26], offer unique characteristics and functionalities. We'll focus on non-interactive VSS schemes, particularly Feldman's VSS scheme, which eliminates the need for further communication between the dealer and users. At Feldman, only the seller sends the message, so that all shareholders can independently verify their shares without further communication. An investor can use the publicly available polynomial to authenticate the received share's accuracy.

3.2. Threshold Cryptography

In the world of public-key cryptography, typically only the owner of a secret key can decrypt messages or sign them, ensuring tight control over sensitive operations. However, there are situations where having this control centralized with one individual or entity isn't ideal. Instead, it can be advantageous to spread this capability across a group of users, with only a subset of them needing to come together to perform decryption or signing tasks. This approach is called threshold cryptography.

On the other hand, the main goal of cryptography is to fend off attackers. Public-key systems, unfortunately, can be vulnerable to attacks where adversaries gain access to the secret key, whether through hacking or insider threats. To address this, systems need to be fortified. Threshold cryptography offers a solution by distributing trust across multiple components or servers. Here, the secret key is split into shares, with each share held by a different server.

the process of generating the key must be decentralized to create shares for each server without relying on a single trusted entity. This decentralization has been achieved in both discrete logarithm and RSA settings. Similarly, for signature schemes, the signing process has also been distributed across servers.

When extending this distribution to the decryption process, similar techniques can be applied, as long as we're only concerned with protecting against chosen-plaintext attacks from passive adversaries. However, for defense against chosen-ciphertext attacks, servers can't start decryption without first confirming the validity of the ciphertext. This precaution is crucial because one of the servers could potentially be compromised by an attacker, who could gain valuable information from attempted invalid ciphertexts. Therefore, it's essential not to delay decryption until the end of the process to verify the servers' ability to decrypt. Instead, a mechanism for publicly verifying the validity of the ciphertext must be integrated. Unfortunately, most known cryptosystems resilient against chosen-ciphertext attacks aren't well-suited for this purpose. This is because in decryption processes, the alleged plaintext is decrypted first, with redundancy checks performed just before returning the plaintext. Since these redundancy checks typically involve hash functions, the final verification can't be efficiently executed in a distributed manner

3.3. Zero Knowledge Proof Method

A ZKP is a method that allows one user (prover) to convince another user (verifier) that the statement is true does not disclose any additional information beyond the truth of the statement. This means the verifier

gains no extra knowledge apart from confirming the truth of the statement. ZKPs are widely used in various blockchain platforms to enhance privacy and scalability. Some popular ZKP algorithms are zkSNARKs [27], zkSTARKs [28], and Bulletproof [29]. These algorithms enable numerical verification, a which basically means that with a particular public and private inputs Ensuring that the program output matches the intended output without having to repeat the whole program allows the verifier instead to build on the truncated proof truth is well emphasized [30].

The robust fidelity of zkSTARKs and bulletproof depends on the Difficulty of the system being validated, while for zkSNARKs, it remains the same regardless of system complexity One disadvantage of About zkSNARKs is that they require a reliable system, which zkSTARKs and Bulletproof don't need. However, zkSNARKs have small proof sizes and consistent verification complexity, making them great for use in smart contract. Moving forward, in this discussion, we concentrate on zkSTARKs.

The 1st successful implementation of zkSTARKs was made possible by StarkNet Libraries [31], which use specific ECCs to verify proofs. Here's how it works: Imagine you have a program. To make it work with zkSTARKs, we convert it into something called an arithmetic circuit. Think of it like transforming the program into a mathematical structure. Triton, which acts like a virtual machine, uses Algebraic Execution Tables and Arithmetic Intermediate Representations as important parts of the process. These help manage the computations efficiently alongside the STARK proof system. The transformed program is then represented in a special format called Quadratic Arithmetic Program (QAP). Checking whether the program gives the right output depends on making sure the solution to the QAP is correct. The reason why zkSTARKs are so cool is that they are both very concise and they keep things private. They are achieved through techniques such as random selection of samples and encryption of data so that mathematical operations can be carried out on them without equal privacy of the contents.

As described in detail in Chapter 5, the proposed threshold cryptography protocol is based on performing ECC functions in the circuit. This requires careful consideration of certain constraints that could potentially impact the feasibility of our solution. A major limitation comes from the fact that the circuit uses modular arithmetic on F_r in a limited range. Consequently, only the ECC defined on F_r can be successfully implemented in the circuit. These curves are crucial for implementing specific cryptographic protocols [32]. Another approach is to design an ECC cycle in which the proof is curve-driven and subsequently proofed in the circuit using F_r

4. Methodology

In this chapter, we develop a framework using a Threshold cryptographic protocol using a smart contract for Distributed Calculation and transmission, and using zkSTARKs to reduce its cost. Next, we dive deeper into the design of the protocol, providing detailed information including steps: *Secret sharing step, proof generation step, secret reconstruction step, verification of share step, and error handling step*.

The implementation of the protocol begins with the *secret sharing step*. In this step, the original secret document is divided into multiple shares using Shamir's Secret Sharing Scheme. Each participant receives a share, ensuring that a subset of shares, equal to or greater than the predefined threshold, is required to reconstruct the secret. Following secret sharing, each participant *proof generation step a zkSTARKs* to prove possession of their share without revealing its value. These zkSTARKs are crucial for maintaining privacy and security during the protocol execution. Once all users have generated their zkSTARKs, the *secret reconstruction process* commences. When enough users come together, typically equal to the threshold, they collectively use Lagrange interpolation to *reconstruct* the original secret document. Subsequently, each

participant independently verifies the validity of their own share and zkSTARKs. If the verification of any share fails, indicating a potential compromise, the protocol *error handles* a halts immediately to prevent further processing with potentially compromised shares, ensuring the integrity and security of the protocol execution.

4.1. Message Model

In this message model, messages are sent and received within a fixed interval called Δ_T, ensuring the continuous synchronization of all users. The blockchain acts as a public platform where messages can be sent and received by the relevant members.

Furthermore, instead of direct communication between individuals, public broadcast channels are used to send messages directly. In this system, the membership consists of two parts - public and private. A public key is shared through the blockchain with all other members. Using this public key with specific information for each round, members create shares for others. This is then applied to a shared encryption scheme that protects their communications over public channels.

4.2. Gas fees Considering Model

Also, it's essential to understand that in a public blockchain system, when you make a transaction, you usually pay for it using the specific cryptocurrency of that blockchain. These fees act as rewards for miners, motivating them to use their computers to perform tasks like creating new blocks and handling transactions. They also help to protect the platform from abuse. Since using computational power has real costs, it deters people from doing harmful or pointless things on the network.

Transaction fees depend on how much work the computer needs to do and how much memory it needs to use. So, transactions that just move money between accounts usually have lower fees than ones that make a smart contract do complicated math or use a lot of memory. Doing operations with elliptic curve cryptography (ECC) is especially expensive. That's why our threshold cryptography uses zkSTARK for do these calculation outside of the main blockchain network.

4.3. *Smart contract Model*

The protocol begins with a crucial step called the trusted setup, which is essential for zkSTARK (explained in chapter 3). This setup process creates the verification proof for the algorithm 4 circuit and for handling errors in algorithm 5 (explained in chapter 4). This system aims to relocate accounts in exchange for valid evidence and to keep the use of invalid parts chain-linked. In a reliable system, a copycat is produced which is commonly referred to as toxic waste. These fraudulent locks should be discarded as they may lead to false evidence [33]. So, using (SMPC) is a smart way to spread trust among many different people or groups.

After completing the reliable configuration task, where the necessary cryptographic parameters are generated and securely shared among the users, the next step involves in zkSDTKG contract where it not only verifies the shares but also it manages rest of the function working proper so that the data will not corrupt Functions such as, Secret Sharing This can share secrets through the Shamir private sharing system. It allows users to create their shares and distribute them securely among other users. (Refer Algorithm 1) Additionally, it includes functions for reconstructing the secret when the required threshold of shares is collected. zkSDTKG Generation: This is done by each participant to obtain ignorant evidence of equity ownership without disclosing the share itself. users can securely generate their ZKPs and submit them to the blockchain. Secret Reconstruction: This is responsible for reconstructing the secret document when the required threshold of users comes together. It uses Lagrange interpolation to combine the valid shares submitted by users and reconstruct the original secret. Verification of Shares: This verifies the validity of each participant's share and their associated ZKP. It ensures that only valid shares are used in the reconstruction process, maintaining the integrity and security of the protocol. Error Handling: This manages

the error handling mechanism of the protocol. It monitors the verification process of shares and ZKP and stops the protocol if any verification fails. This ensures that no further processing occurs with potentially compromised shares, protecting the confidentiality and reliability of the secret reconstruction process.

The zkSDTKG contract should be configured with various settings, such as the address of the verification contract, thresholds, deadlines for different phases, curve parameters, and any other information needed to dynamically prepare part sets.

4.4. User Involvement

After successfully deploying the zkSDTKG contract, various members can now initiate the process to become qualified users. Unlike traditional threshold cryptography protocols that operate without a blockchain, we have several flexible options to assemble the user group dynamically. We aim to avoid restricting ourselves to a particular method. In this context, the zkSDTKG contract establishes and rigorously enforces the eligibility criteria for participating in the protocol execution. In a controlled environment, imagine a situation where a special type of cryptographic contract zkSDTKG, sets the rule that only its creator has the power to decide who can join. This creator would need to provide a list of approved users to the smart contract in advance. Then, the zkSDTKG contract would check if a person seeking to join is on the approved list. If they are, the contract would allow them to join the group of users. However, there's a significant problem here. The creator of the contract acts as a central authority, which goes against the idea of decentralization in the system.

Other unrestricted methods may permit anyone to participate without needing a central authority. For instance, the zkSDTKG contract could employ a basic system where anyone can join until a set number of users is reached, provided they put up some tokens as security. However, this method isn't feasible in most situations because it lacks Sybil resistance, meaning individuals could register with more than one identities [34]. In a worst-case scenario, a malicious individual could potentially create enough fake identities to gain complete control and access to the key. Another idea is to use the amount of tokens users hold not just as collateral but also as a factor in deciding who can participate, similar to systems that rely on Proof of Stake (PoS). For instance, in the zkSDTKG contract, only the top parties with the most tokens deposited are allowed to join. If users act wrongly during the protocol, they might lose some of their tokens as a penalty. This setup encourages honesty and makes it harder for one user to create multiple identities by staking a lot of tokens. Another variation of this idea would allow regular token holders to donate their tokens to vote as users for some candidates. This not only makes the system more secure by using Proof of Stake mechanisms but also adds a level of decentralization and trust through voting processes.

4.5.TC with smart contract using zk -STARKs

Once we've finished all the necessary preparatory steps for carrying out the protocol, we'll now detail its description and specifications (see Protocol 1). This section involves a thorough look at the secret sharing, proof generation phase, secret reconstruction phase,verification of shares and error handling phase explaining each step in a clear and logical order with an form of example is explained.

4.5.1. Secret sharing step

Protocol 1 of step 2, let's follow Alice as she employs Shamir private sharing system to securely distribute a document with the value of 42 to Bob and Charlie. Initially, Alice establishes the parameters for the sharing process, selecting three users: herself, Bob, and Charlie. She then chooses a prime number, 67, larger than the value of the document. To create shares, Alice generates a polynomial $f(x) = 5x^2 + 3x + 42$, randomly assigning coefficients.

Next, Alice calculates shares by substituting x = 1, 2, and 3 into the polynomial. These calculations yield results of 50, 95, and 168, respectively. To ensure the shares are within the confines of the chosen prime,

Alice reduces each result modulo 67, resulting in shares of 50, 28, and 34. She then allocates each participant a specific share: Alice holds 50, Bob receives 28, and Charlie is entrusted with 34.

For distribution, Alice securely transmits Bob's share to him and Charlie's share to Charlie, ensuring each participant only receives their designated share. Consequently, Bob and Charlie, even if they were to collaborate, cannot ascertain the contents of the secret document without Alice's share.

4.5.2. zkSDTKG for zk-STARKs Generation Step (π_i)

After completing secret sharing step, Alice, Bob, and Charlie took the next step by generating proofs with the assistance of the zkSDTKG Contract (see protocol 1 step 3). This contract allowed them to verify their possession of valid shares without revealing the actual values of those shares. Each participant, including Alice, Bob, and Charlie, independently created their proof. For example, Alice crafted her proof using her share, denoted as $share_A$, which in her case was 50. She formed a commitment to her share, possibly by creating a cryptographic hash of $share_A$and then crafted a proof, denoted as $proof_A$, demonstrating her knowledge of $share_A$ without disclosing its numerical value. Bob and Charlie followed identical procedures, generating their own inputs, commitments, and proofs.

After the zkSDTKG Contract generation phase, each participant securely transmitted their proof components to the other users involved in the secret sharing. This ensured that each party possessed the necessary information to verify the validity of each other's shares during the reconstruction phase of the secret document, while still maintaining the confidentiality of their own share. By securely exchanging their proof components, Alice, Bob, and Charlie established a robust framework where they could collaborate without divulging sensitive information, thus upholding the privacy and integrity of the shared secret. This process enhanced the overall security of their collaborative efforts, fostering trust and confidence among the users.

4.5.3. Secret Reconstruction step:

In the secret reconstruction step using Lagrange interpolation, the goal is to reconstruct the secret document D from the shares and proof provided by the users. In this scenario, users Alice, Bob, and Charlie hold shares of the secret: $share_A = 50$, $share_B = 28$, and $share_C = 34$ respectively, along with their corresponding proof. When at least $t = 20$ users come together, the secret document can be reconstructed.

(See protocol 1 of step 4), Lagrange interpolation is employed to compute the secret document D using shared points among users, denoted as $x = 1,2,3$.

The formula for reconstructing D is:

$$D = \sum_{i=1}^{t} \frac{\Pi_{j \neq i}(x - x_j)}{\Pi_{j \neq i}(x_i - x_j)} . y_j \quad (1)$$

For each shared point x the formula is evaluated. For instance, when $x = 1$, the calculation involves each participant's share:

$$D = \frac{(1-2)(1-3)}{(1-2)(1-3)} . 50 + \frac{(1-1)(1-3)}{(2-1)(2-3)} . 28 + \frac{(1-1)(1-2)}{(3-1)(3-2)} . 34$$

Similarly, calculations are performed for $x = 2$ and $x = 3$ to obtain D at each point (see protocol 1 step 4).

Once the values of D for each x are computed, the secret document is successfully reconstructed, ensuring that the confidentiality and integrity of the document are maintained throughout the process.

4.5.4. Verification of Shares step:

In the process of verifying shares in the cryptographic reconstruction protocol, each participant undertakes a crucial step to ensure the integrity and authenticity of their own share. This verification process is pivotal for maintaining the security of the overall reconstruction process (see protocol 1 step 5). Let's delve into how each participant verifies their respective share and zk-STARKs to ascertain their validity.

Firstly, Alice, as one of the users, conducts her verification by assessing the validity of her zkSTARKs. She does so by invoking a verification function that takes her share, commitment to her share, and her zkSTARKs as inputs. If this verification function returns true, it signifies that Alice's ZKP is indeed valid, thereby corroborating the authenticity of her share. Conversely, if the verification process fails and the function returns false, it indicates a discrepancy or potential compromise in Alice's zkSTARKs, prompting further investigation.

Similarly, Bob, another participant in the reconstruction protocol, follows suit by verifying the validity of his zk-STARKs. Employing the same approach as Alice, Bob evaluates whether his share, commitment to his share, and his zk-STARKs align to pass the verification process. A successful outcome, where the verification function returns true, affirms the integrity of Bob's zk-STARKs and, by extension, his share. Conversely, if the verification process yields a false result, Bob must address any discrepancies or potential security risks associated with his zk-STARKs.

Lastly, Charlie, the third participant, carries out the verification of his zk-STARKs to validate the authenticity of his share. Utilizing a similar methodology to Alice and Bob, Charlie submits his share, commitment to his share, and his zk-STARKs to the verification function. Upon receiving a true outcome from the verification process, Charlie can be assured of the legitimacy of his zk-STARKs and, consequently, his share. However, if the verification process returns false, it signifies a potential anomaly or security breach in Charlie's zk-STARKs, necessitating immediate attention.

Only when all users' verifications return true can the reconstruction process proceed with confidence. Any instance of a false verification result warrants halting the protocol to mitigate the risk of compromised shares and maintain the security of the cryptographic reconstruction endeavour.

4.5.5. Error Handling step:

In the process we've been discussing, after Alice, Bob, and Charlie have completed the initial setup by verifying the validity of their shares and zk-STARKs, the next crucial step is error handling. This step ensures the integrity of the protocol by promptly addressing any issues that may arise during verification.

During error handling, if any participant's share fails verification, the protocol halts immediately. For instance, let's consider a scenario where Bob's zk-STARKs verification fails. This failure indicates a potential issue with the integrity or accuracy of Bob's share. As a result, the protocol stops to prevent further processing with potentially compromised data (see protocol 1 of step 6).

All users are promptly notified of the halt in the protocol due to the verification failure of Bob's share. This transparency ensures that everyone involved is aware of the situation and can collectively address it.

Halting the protocol in response to verification failures is crucial for maintaining the security and integrity of the process. It prevents the propagation of potentially incorrect or compromised data further down the line. Following the halt, users can investigate the cause of the verification failure and take appropriate actions. This may include re-verifying Bob's share or replacing it with a valid one if necessary, ensuring the continuation of the protocol with accurate and reliable data.

Protocol 1 of Threshold Decryption Protocol with example

Suppose Alice wants to share a secret document $D = 42$ with Bob and Charlie using Shamir's Secret Sharing Scheme.

1) Setup:
    a. $n = 3$ users: Alice, Bob, and Charlie.
    b. Choose a prime number $p > D$, let's say $p = 67$.
    c. Randomly choose coefficients for the polynomial $f(x)$: Let's say $f(x) = 5x^2 + 3x + 42$.
    d. Generate shares using Shamir's SSS.

2) Secret Sharing Step:
    a. $f(0) = \sum_{i=1}^{t} a_i \,.\, x^i \; mod \; p$
    b. According to Algorithm 1 corresponding share need to be substituted $x = 1,2,3$ into the polynomial $f(x)$:
        i. $f(x) = 5(1)^2 + 3(1) + 42 = 50$
        ii. $f(x) = 5(2)^2 + 3(2) + 42 = 95$
        iii. $f(x) = 5(3)^2 + 3(3) + 42 = 168$
    c. Reduce each result modulo p:
        i. $50 \,\%\, 67 = 50$
        ii. $95 \,\%\, 67 = 28$
        iii. $168 \,\%\, 68 = 34$
    d. Assign each participant a share:
        i. Alice: Share $share_A = 50$
        ii. Bob: Share $share_B = 28$
        iii. Charlie: Share $share_C = 34$
    e. Distribution:
        i. Alice securely sends Share $share_A$ to Bob and Share $share_c$ to Charlie.
        ii. Bob receives Share $share_A$
        iii. Charlie receives Share $share_c$
3) zkSDTKG for zk-STARKs Generation Step (π_i):
    a. In algorithm 2, each users will generate their proof. $\pi_i = (input_i, commit(y_i), proof_i)$
    b. Alice:
        i. $input_A : share_A = 50$
        ii. $Commitment_A = hash(50)$
        iii. $proof_A$:$zkSDTKG(share_A)$
    c. Bob:
        i. $input_B : share_B = 28$
        ii. $Commitment_B = hash(28)$
        iii. $proof_B$:$zkSDTKG(share_B)$
    d. Charlie:
        i. $input_c : share_c = 34$
        ii. $Commitment_c = hash(34)$
        iii. $proof_c$:$zkSDTKG(share_c)$
    e. Distribution:
        i. Alice securely sends $(Commitment_A, proof_A)$ to Bob and Charlie.
        ii. Bob securely sends $(Commitment_B, proof_B)$ to Alice and Charlie.
        iii. Charlie securely sends $(Commitment_C, proof_C)$ to Alice and Bob.
4) Secret Reconstruction Step:
    a. Alice, Bob, and Charlie have their shares: $share_A = 50$, $share_B = 28$, $share_C = 34$
    b. They also have generated their respective proof: π_A, π_B, and π_C.
    c. When enough users come together (in this case, at least $n = 2$ users), they use Lagrange interpolation to reconstruct the secret document D.
    d. Using the shares and zk-STARKs provided by Alice, Bob, and Charlie, the reconstruction can be done as follows:
    e. Let's assume $x = 1,2,3$ are the points shared among users.
    f. The secret document D can be reconstructed using Lagrange interpolation $\{D = \sum_{i=1}^{t} \frac{\Pi_{j \neq i}(x - x_j)}{\Pi_{j \neq i}(x_i - x_j)} \,.\, y_j\}$
    g. For $x = 1$

i. $D = \frac{(1-2)(1-3)}{(1-2)(1-3)}.50 + \frac{(1-1)(1-3)}{(2-1)(2-3)}.28 + \frac{(1-1)(1-2)}{(3-1)(3-2)}.34$

h. For $x = 2$

i. $D = \frac{(2-2)(2-3)}{(1-2)(1-3)}.50 + \frac{(2-1)(3-3)}{(2-1)(2-3)}.28 + \frac{(2-1)(2-2)}{(3-1)(3-2)}.34$

i. For $x = 2$

i. $D = \frac{(3-2)(3-3)}{(1-2)(1-3)}.50 + \frac{(3-1)(3-3)}{(2-1)(2-3)}.28 + \frac{(3-1)(3-2)}{(3-1)(3-2)}.34$

j. By determine the values of D for each x the secret document can be reconstructed using Algorithm 3.

5) Verification of Shares Step:
   a. From Step 4 Alice, Bob, and Charlie have reconstructed the secret document D using Lagrange interpolation.
   b. Each participant also has their own share and proof:
      i. Alice: $share_A : 50, \pi_A$
      ii. Bob: $share_B : 28, \pi_B$
      iii. Charlie: $share_C : 34, \pi_C$
   c. In Algorithm 4, each participant confirms the accuracy of their own portion, and Zero-Knowledge Proof (ZKP) is utilized for verification:
      i. Alice verifies (π_A):
         1. Alice checks if $verify[share_A, commit(share_A), proof_A]$ returns true.
         2. If it returns true, Alice's ZKP is valid. Otherwise, it fails.
      ii. Bob verifies (π_B):
         1. Bob checks if $verify[share_B, commit(share_B), proof_B]$ returns true.
         2. If it returns true, Bob's ZKP is valid. Otherwise, it fails.
      iii. Charlie verifies (π_C):
         1. Charlie checks if $verify(share_C, commit(share_C), proof_C)$ returns true.
         2. If it returns true, Charlie's zk-STARKs are valid. Otherwise, it fails.
   d. Result
      i. If all users' verifications return true, indicating that all zk-STARKs are valid, the reconstruction process can proceed.
      ii. If any participant's verification returns false, indicating an invalid zk-STARKs, the verification fails, and the protocol should halt and move to step 6.

6) Error handling Step:
   a. Alice, Bob, and Charlie have reconstructed the secret document D using Lagrange interpolation.
   b. Each participant has their own share and zk-STARKs, and they have verified their shares.
   c. If any participant's verification fails (i.e., their zk-STARKs is invalid), the protocol halts.
   d. Upon halting, the users should go back to the secret reconstruction step to repeat the process with valid shares.
   e. If Bob's verification fails, meaning that the proof of knowledge he provided is invalid, the protocol should stop, as described in algorithm 5. Then, Alice, Bob, and Charlie need to start over the reconstruction process following the steps 4 and 5 shown in protocol 1.
   f. Reconstruction Retry

i. Alice, Bob, and Charlie repeat the secret reconstruction step with new shares and zk-STARKs.
ii. After generating new shares and zk-STARKs, they verify their shares again before proceeding with the reconstruction.

g. Result
i. If all users' verifications pass, they can continue with the reconstruction process
ii. Otherwise, they may need to repeat the process until all shares are verified successfully.

Algorithm 1 – Function for Secret sharing

```
Function splitSecret (secret, num_shares, threshold, prime):
coefficients = [secret] + [random.randint (0, prime - 1) for _ in range (threshold - 1)]
shares = []
for i in range(num_shares):
x = i + 1
share = sum (coeff * x**power for power, coeff in enumerate(coefficients)) % prime
shares.append ((x, share))
return shares
```

Algorithm 2 – Function for zkSDTKG for zk-STARKs Generation

```
Function generateProof (share, prime):
x, y = share
r = random.randint(1, prime - 1)
commitment = pow (r, 2, prime)
proof = pow (y, 2, prime)
return (x, commitment, proof)
```

Algorithm 3 - Secret reconstruction using Lagrange interpolation

```
Function reconstructSecret (shares, prime):
secret = 0
for i in range(len(shares)):
xi, yi = shares[i]
num, den = 1, 1
for j in range(len(shares)):
if i != j:
    xj, _ = shares[j]
    num = (num * (-xj)) % prime
    den = (den * (xi - xj)) % prime
secret = (secret + (yi * num * pow (den, -1, prime))) % prime
return secret
```

Algorithm 4 – Function for Verification of Shares

```
Function zkVerifier (share, prime):
x, commitment, proof = share
return pow (x, 2, prime) == (commitment * proof) % prime
```

Algorithm 5 – Function for Error handling

```
Function handleError ():
```

2. Print ("Error: Verification of shares failed. Halting the protocol.")
3. Exit ()

5. Prototype implementation

In this part, we provide more details about how the client software is created and the smart contract are developed. You can find both of these as open-source software on GitHub[1], meaning they are publicly available for anyone to view and use.

5.1. Smart Contracts

We've successfully design smart contract using Solidity programming language for Polygon blockchain, which is one of the top platforms for smart contract today. Polygon offers a wide range of tools and strong support for developers. Notably, it provides pre-made contracts for important cryptographic tasks like elliptic curve operations and pairing checks on the BLS12-381 curve [35]. These are crucial for verifying zk-STARK proofs. Our prototype, designed specifically for Polygon blockchain, smoothly integrates with other blockchain platforms because of its flexible architecture. This flexibility also allows it to work with other blockchain ecosystems, as long as they have the necessary smart contract features and support for zk-STARK verification. This adaptable solution demonstrates our commitment to making our technology work across different blockchain systems, while still meeting the high security standards of zk-STARK technology.

We've decided to go with the simple registration process we discussed earlier in Chapter 4 because it meets our basic implementation requirements well. This decision allows us to concentrate on the critical aspects of the protocol without adding unnecessary complications. The zkSDTKG contract lets users sign up to a set limit and mandates reclaim submission. Afterward, users can get back their reclaim from the zkSDTKG contract once the protocol is done, as long as they've been honest throughout.

The threshold cryptography used to depend on the number of registered users, denoted by n. In the zkSDTKG contract, the computation works like this: $[(n + 1)/2]$. This indicates that a majority of the individuals who have signed up are required to collaborate in order to reconstruct the distributed private key. Additionally, it's specified that at least $[(2n + 1)/3]$ users must efficiently convey their secrets for facilitating the submit of the public key. This ensures the system can keep functioning even if some users behave maliciously. These requirements represent the minimum needed based on relevant literature (see Chapter 2 for more details).

The zkSDTKG contract is designed to save storage space on the blockchain by cleverly minimizing the direct storage of participant data. Instead of storing all the details of user participation and commitments directly, it stores the archive of the smart contract only with the most important information about the first proposal for each user. The rest of the data, including share and Pledges, are stored in the call-data, with only a compact Keccak-256 hash stored in the contract. This approach is more cost-effective and efficient. However, Ethereum currently lacks efficient implementations of hash functions specifically tailored for arithmetic, which would greatly improve efficiency when computing hashes within a circuit. Examples of such hash functions include MiMC [36] or Rescue-Prime [37]. To keep users informed about recent distributions, the zkSDTKG contract sends out an event. This event notifies all users about distributions made by specific users, ensuring transparency and visibility into the distributed shares and commitments.

The zkSDTKG contract didn't require any manual setup. Triton VM, as described in Chapter 5, can automatically create verification contracts for the circuits involved. These contracts, created by Triton VM,

[1] https://github.com/RahulTharammal/Threshold-Cryptography-with-smart-contract-using-zk-STARKs

use Ethereum's pre-made contracts to ensure efficient verification of proofs at a reasonable cost.

5.2 Guest Software

The user software, model using the Rust programming language, supports the smart contracts by employing an offchain component responsible for implementing the zkSDTKG contract to generate a shared private key This software integrates with the Triton VM [38] toolbox for generating zkSTARKs, leveraging its Algebraic Execution and Arithmetic Intermediate components to support the zkSTARK proof system on the Ethereum platform. Users can use the Triton VM toolbox to calculate witnesses for a given circuit and generate proofs of their preferred authentication protocol. For blockchain-based proof verification, RISC Zero has been selected as it offers a more cost-effective solution.

Our implementation provides the computational intermediate level of the Triton VM bound by Algorithms 1 through 5. In addition, we have integrated the required functions from the standard library of the Triton VM as our model for ECC. Importantly, our protocol is adaptive, allowing compatibility with other computational methods as long as their base location coincides with the digital location of the stations used for zkSTARKs This flexibility ensures that our protocol can operate efficiently in computer systems or preferences without honesty or diligence

To overcome the limitations of the Triton VM, the user uses Rust language bindings in the Polygon smart contract to execute the commands to estimate the tenant and verify for Algorithms 2 & 4 and then zkVerifier or the generateProof functions.

6. Protocol Assessment

Presenting the implementation of the protocol, this section examines the estimation of the protocol based on cost, throughput, and memory usage. This analysis provides valuable insight into the additional burden created by the smart contract and the resources required to verify the zkVerifier and generateProof tasks.

In these iterations, we carefully record gas consumed, proof generating time, and memory used as we repeat the process five consecutive times before gradually increasing the number of users increased, and we doubled them each time to avoid reaching all users 150 This leaves us with 40 rounds of action. We conduct our analysis on an Amazon EC2 instance with a specific instruction: this AMI uses a 64-bit architecture, 1 virtual CPU, and a z-series instance. The device has 8 gigabytes of DDR4 RAM at 2666 MHz and an SSD with a maximum capacity of 128 megabytes per second. In addition, we configured a local blockchain instance using Hardhat network version 2.20.1 to facilitate the implementation and execution of smart contracts.

6.1. Gas Consumption

When we execute smart contracts on the polygon network, we rely on a system called gas to ensure that the network resources are used efficiently and fairly. Gas acts as a safeguard against excessive resource consumption and also acts as an incentive for miners who dedicate their computational power to validating transactions. This principle isn't unique to polygon; other platforms hosting smart contracts employ similar mechanisms to compensate miners. Now, let's delve into the expenses associated with running smart contracts, particularly focusing on the gas utilized by functions such as splitSecret, generateProof, reconstructSecret, zkVerifier and handleError.

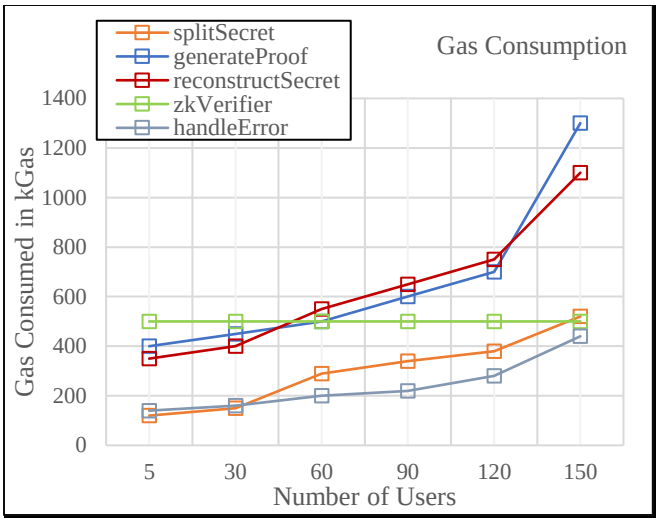

Figure 1: Gas Consumed by each Function

The data presented in Figure 1 illustrates that both the splitSecret and handleError functions exhibit minimal gas usage. Specifically, splitSecret's gas consumption averages 120 kGas for 5 users, rising to 520 kGas for 150 users. These costs remain low due to the inclusion of shares and commitments in the calldata, while only hash values are stored in the zkSDTKG contract. Similarly, the handleError function demonstrates even lower gas usage, with 140 kGas for 5 users and 440 kGas for 150 users.

Furthermore, it is clear that the gas consumption in the *zkVerifier* project is still flat at 500 kGas, despite the number of users involved. This inconsistency exists because the verify function only requires zkSTARKs with three ECC points as input. zkSDTKG contract do not perform any actions based on the number of users and resolves *handleError* immediately if the authentication was successful. However, if more *handleErrors* are generated, gas consumption increases slightly. This happens because the smart contract must go through all the inputs to find a matching index.

Certainly, as more people participate, gas consumption increases during the *generateProof* process. For example, it is 400 kilograms of gas for 5 people and jumps to 1300 kilograms of gas for 150. This increase occurs because the zkSDTKG contract requires computing the hash of all initial parameters to verify the zkSTARKs. In addition, the study shows that the execution of the *reconstructSecret* project consumes 350 kGas of gas for 5 people but jumps to 1100 kGas of gas for 150 people the time has passed is enough. This is because the smart contract must remove the node before generating the public key. However, since such events are rare, they do not significantly affect gas consumption.

6.2. Throughput Performance of generating proof

While smart contract come with their own set of complexities, integrating zk-STARKs for executing certain functions like justification and key derivation adds another layer of intricacy. Although this approach can help save costs in executing smart contracts, it also demands users to allocate more local resources for generating proofs. An interesting consideration is how the time taken by users to generate these proofs changes as the number of users increases.

The information in Figure 2 shows a clear connection between the number of users and the time spent on the two tasks: *generateProof* and *reconstructSecret*. Interestingly, while both functions show a linear relationship with user count, the time required for *reconstructSecret* rises more steeply compared to *generateProof*. This difference in escalation is because the time it takes for key derivation is directly tied to the number of users, impacting *reconstructSecret* significantly. On the other hand, *generateProof's* duration

relies more on the threshold value. For instance, with 5 users, *generateProof* takes around 20 seconds, while *reconstructSecret* takes 30 seconds. However, with 150 users, these times significantly increase to roughly 350 and 508 seconds, respectively.

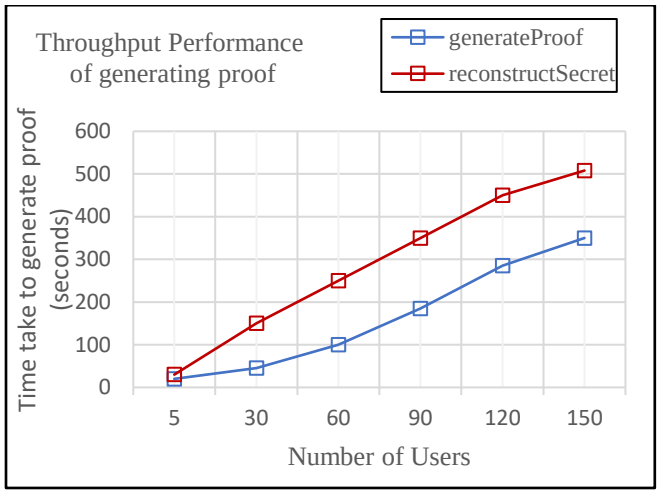

Figure 2: Time taken for each proof generated

While the proof generated time is typically not a major concern within blockchain systems as these networks are not primarily utilized for time-lapse purposes, it should be carefully considered when determining when the zkSDTKG contract expires. This is particularly crucial to ensure that users have adequate time for proof generation, especially for the generateProof function. Furthermore, there are numerous opportunities for optimization aimed at reducing proof generation time, including the utilization of hash functions tailored for Arithmetic and the enabling of multi-threading for Triton VM.

6.3. Memory usage

Furthermore, our examination also looks into how memory usage is affected when generating proofs. Memory consumption is often a major worry, especially when dealing with the inefficiencies of specific calculations, like keccak_256 in zk-STARKs. This investigation also considers how memory usage changes with varying numbers of users.

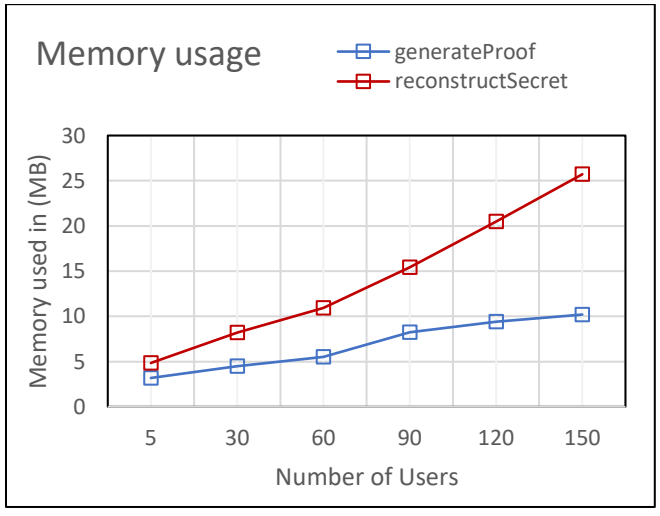

Figure 3: Memory used by each function while generating Proof

The memory usage, as shown in Figure 3, consistently increases as the number of users increases, which is similar to what we observed in the time it takes to generate proofs. For example, when we generate a proof using the function generateProof, the amount of memory needed goes up from 3.19 gigabytes with 5 users to 10.2 gigabytes with 150 users. Likewise, when we use the function reconstructSecret, the memory usage starts at 4.85 gigabytes for 5 users and goes up to 25.715 gigabytes for 150 users. This means that more memory is needed as the number of users grows, making it harder for new users to join without having sufficient resources for proof generation. There is a chance that a custom hash function is provided for arithmetic operations to optimize memory usage.

7. Conclusion

In our research, we emphasize the important role of threshold cryptography in building secure systems, especially within the realm of blockchain technology. We argue that blockchains offer a unique opportunity to advance threshold cryptosystems due to their decentralized nature, which allows for distributed computation and communication. To make the most of this potential, we introduce a new method for implementing threshold cryptography that seamlessly incorporates blockchain technology. Using smart contracts, our method allows for flexible participation from multiple parties, enhances security by offering incentives, ensures the correctness of protocol execution, and establishes a public communication channel through the blockchain. Additionally, our method utilizes zk-STARKs to justify actions and derive keys, enabling complex computations to be done off-chain to reduce costs. We've put our method into practice on the Ethereum platform and conducted thorough evaluations to demonstrate its practicality and efficiency in terms of costs, performance, and memory usage. Going forward, our future research will focus on further reducing the costs associated with smart contract and adding enhancements from other protocols to further increase the performance and scalability of our approach.

References

[1]. Y. Desmedt, "Threshold cryptography," *Transactions on Emerging Telecommunications Technologie*s, vol. 5, no. 4, pp. 449–458, Jul. 1994, doi: 10.1002/ett.4460050407.

[2]. S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," Scientific Research Publishing. [Online]. Available: https://www.scirp.org/reference/ReferencesPapers?ReferenceID=1522950

[3]. H.-N. Dai, Z. Zheng and Y. Zhang, "Blockchain for Internet of Things: A Survey," IEEE Internet of Things Journal, vol. 6, no. 5, pp. 8076–8094, Oct. 2019, doi: 10.1109/JIOT.2019.2920987.

[4]. M. B. Mollah et al., "Blockchain for Future Smart Grid: A Comprehensive Survey," IEEE Internet of Things Journal, vol. 8, no. 1, pp. 18–43, Jan. 2021, doi: 10.1109/JIOT.2020.2993601.

[5]. D. Boneh and M. K. Franklin, "Identity-Based Encryption from the Weil Pairing," in Lecture Notes in Computer Science, 2001, pp. 213–229, doi: 10.1007/3-540-44647-8_13.

[6]. B. Körbel, M. Sigwart, P. Frauenthaler, M. Sober, and S. Schulte, "Blockchain-based result verification for computation offloading," arXiv.org, Oct. 21, 2021. [Online]. Available: https://arxiv.org/abs/2110.11090

[7]. L. Chen, K. J. Harrison, D. Soldera, and N. P. Smart, "Applications of multiple trust authorities in pairing based cryptosystems," in *Lecture Notes in Computer Science*, 2002, pp. 260–275. doi: 10.1007/3-540-45831-x_18.

[8]. S. S. Al-Riyami and K. G. Paterson, "Certificateless public key cryptography," in *Lecture Notes in Computer Science*, 2003, pp. 452–473. doi: 10.1007/978-3-540-40061-5_29.

[9]. D. Pointcheval, "Practical Security in Public-Key Cryptography," in *Lecture Notes in Computer Science*, 2002, pp. 1–17. doi: 10.1007/3-540-45861-1_1.

[10]. R. Canetti and S. Goldwasser, "An Efficient threshold Public Key Cryptosystem Secure Against Adaptive Chosen Ciphertext Attack (Extended Abstract)," in *Lecture Notes in Computer Science*, 1999, pp. 90–106. doi: 10.1007/3-540-48910-x_7.

[11]. V. Shoup and R. Gennaro, "Securing threshold cryptosystems against chosen ciphertext attack," in *Lecture Notes in Computer Science*, 1998, pp. 1–16. doi: 10.1007/bfb0054113.

[12]. P.-A. Fouque and D. Pointcheval, "Threshold Cryptosystems Secure against Chosen-Ciphertext Attacks," in *Lecture Notes in Computer Science*, 2001, pp. 351–368. doi: 10.1007/3-540-45682-1_21.

[13]. T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Springer eBooks*, 2007, pp. 10–18. doi: 10.1007/3-540-39568-7_2.

[14]. M. Bellare and P. Rogaway, *Random oracles are practical*. 1993. doi: 10.1145/168588.168596.

[15]. C.-P. Schnorr, *Efficient Identification and Signatures for Smart Cards (Abstract)*. 1989, pp. 688–689. [Online]. Available: https://dblp.uni-trier.de/db/conf/eurocrypt/eurocrypt89.html#Schnorr89

[16]. D. Pointcheval and J. Stern, "Security arguments for digital signatures and blind signatures," *Journal of Cryptology*, vol. 13, no. 3, pp. 361–396, Mar. 2000, doi: 10.1007/s001450010003.

[17]. C.-P. Schnorr and M. Jakobsson, "Security of signed ElGamal encryption," in *Lecture Notes in Computer Science*, 2000, pp. 73–89. doi: 10.1007/3-540-44448-3_7.

[18]. P. Thorncharoensri, W. Susilo, and Y. Mu, "Identity-Based Identification Scheme Secure against Concurrent-Reset Attacks without Random Oracles," in *Lecture Notes in Computer Science*, 2009, pp. 94–108. doi: 10.1007/978-3-642-10838-9_8.

[19]. M. Naor and M. Yung, "Public-key cryptosystems provably secure against chosen ciphertext attacks," Jan. 01, 1990. doi: 10.1145/100216.100273.

[20]. A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979, doi: 10.1145/359168.359176.

[21]. J. Katz and Y. Lindell, *Introduction to modern Cryptography*. 2020. doi: 10.1201/9781351133036.

[22]. B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, *Verifiable secret sharing and achieving simultaneity in the presence of faults*. 1985. doi: 10.1109/sfcs.1985.64.

[23]. P. Feldman, A practical scheme for non-interactive verifiable secret sharing. 1987. doi: 10.1109/sfcs.1987.4.

[24]. T. Pedersen, "Non-Interactive and Information-Theoretic secure verifiable secret sharing," in *Springer eBooks*, 2007, pp. 129–140. doi: 10.1007/3-540-46766-1_9.

[25]. T. Rabin and M. Ben-Or, *Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract)*. 1989, pp. 73–85. [Online]. Available: https://doi.org/10.1145/73007.73014

[26]. B. Schoenmakers, "A simple publicly verifiable secret sharing scheme and its application to electronic voting," in *Lecture Notes in Computer Science*, 1999, pp. 148–164. doi: 10.1007/3-540-48405-1_10.

[27]. E. Ben–Sasson, A. Chiesa, E. Tromer, and M. Virza, *Succinct non-interactive zero knowledge for a von Neumann architecture*. 2014, pp. 781–796.

[28]. E. Ben–Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable Zero Knowledge with No Trusted Setup," in *Lecture Notes in Computer Science*, 2019, pp. 701–732. doi: 10.1007/978-3-030-26954-8_23.

[29]. B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, *Bulletproofs: Short Proofs for Confidential Transactions and More*. 2018. doi: 10.1109/sp.2018.00020.

[30]. A. E. Kosba, D. Papadopoulos, C. Papamanthou, and D. Song, "MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs.," *IACR Cryptology ePrint Archive*, vol. 2020, p. 278, Jan. 2020, [Online]. Available: https://eprint.iacr.org/2020/278.pdf

[31]. "Starknet documentation." https://docs.starknet.io/documentation/quick_start/environment_setup/

[32]. M. Bellés-Muñoz, B. Whitehat, J. Baylina, V. Daza, and J. L. Muñoz-Tapia, "Twisted edwards elliptic curves for Zero-Knowledge circuits," *Mathematics*, vol. 9, no. 23, p. 3022, Nov. 2021, doi: 10.3390/math9233022.

[33]. S. Bowe, A. Gabizon, and M. D. Green, "A multi-party protocol for constructing the public parameters of the Pinocchio ZK-SNARK," in *Lecture Notes in Computer Science*, 2019, pp. 64–77. doi: 10.1007/978-3-662-58820-8_5.

[34]. J. R. Douceur, "The Sybil attack," in *Lecture Notes in Computer Science*, 2002, pp. 251–260. doi: 10.1007/3-540-45748-8_24.

[35]. "BLS12-381 for the rest of us - HackMD," *HackMD*. https://hackmd.io/@benjaminion/bls12-381

[36]. M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity," in *Lecture Notes in Computer Science*, 2016, pp. 191–219. doi: 10.1007/978-3-662-53887-6_7.

[37]. A. Szepieniec, T. Ashur, and S. Dhooghe, "Rescue-Prime: a Standard Specification (SoK).," *IACR Cryptology ePrint Archive*, vol. 2020, p. 1143, Jan. 2020, [Online]. Available: https://eprint.iacr.org/2020/1143.pdf

[38]. "TritonVM/triton-vm: Triton is a virtual machine that comes with Algebraic Execution Tables (AET) and Arithmetic Intermediate Representations (AIR) for use in combination with a STARK proof system.," *GitHub*. https://github.com/TritonVM/triton-vm (accessed on March 10,2024)