

DECentralized oracles (DECO): Extensions and Applications

Wenxing Duan



Master of Science
School of Informatics
University of Edinburgh
2024

Abstract

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Wenxing Duan)

Acknowledgements

Any acknowledgements go here.

Table of Contents

1	Introduction	1
2	Background	4
2.1	Blockchain & Smart Contract	4
2.2	Oracle	5
2.3	Zero Knowledge Proof	5
2.4	Secure Multi-Party Computation	6
2.5	TLS	7
3	DECentralized oracles (DECO)	10
3.1	Three-Party Handshake	10
3.2	Query Execution	15
3.3	Proof Generation	16
4	Practical Improvement	19
4.1	Motivation	19
4.2	Direct Approach	20
4.3	Proof of Integrity on MAC Tag	22
4.4	Proof of Integrity on Merkle Trees	25
4.5	Proof of Integrity on AES Encryption	30
4.6	Comparison	33
4.7	Further Privacy Improvement	35
4.7.1	Components Composability	35
4.7.2	Unlinkability	36
5	Security	38
5.1	Completeness	38
5.2	Soundness	38

5.3 Blindness	38
6 Limitation & Conclusion	39
Bibliography	40

Chapter 1

Introduction

In 2008, Satoshi Nakamoto introduced Bitcoin through the publication of "Bitcoin: A Peer-to-Peer Electronic Cash System," which marked the advent of blockchain technology [8]. By 2014, Ethereum [2] was developed, enhancing blockchain capabilities from merely processing transactions (Blockchain 1.0) to functioning as a "world computer" with the addition of Turing Complete smart contracts (Blockchain 2.0) [2]. This evolution expanded blockchain's potential, enabling the execution of code on the blockchain.

The inherent security and privacy features of blockchain wallets have catalyzed the rapid growth of decentralized finance (De-Fi). Platforms built on blockchains like Ethereum leverage smart contracts to provide financial services such as loans and exchanges, with approximately \$27 billion currently invested in cryptocurrency. Securing these funds is a critical focus for blockchain researchers. Moreover, the structure of blockchain limits smart contracts from accessing external internet data, necessitating the use of oracles for data provision. Ensuring the accuracy of oracle-provided data is vital for financial safety in De-Fi and for functions like decentralized identity authentication in systems such as CanDiD [7]. Consequently, enhancing the speed, cost-efficiency, and accuracy of oracle services is a key research priority in the blockchain field.

One notable development in this area is DECO (DECentralized oracle) [10], a blockchain oracle protocol developed by Cornell University researchers in collaboration with Chainlink. DECO is designed for decentralized applications (DApps) and offers a secure, privacy-preserving method to utilize internet data. Its core functionality employs zero-knowledge proofs (ZKP) to ensure data privacy and integrity within smart contracts. DECO allows smart contracts to authenticate specific information without exposing

sensitive data such as usernames and passwords, which is crucial for protecting user privacy. By verifying proof generated through a three-party TLS handshake protocol, DECO enables smart contracts to authenticate data without the willing involvement of data providers, involving a Verifier as a third party. Specifically, the DECO protocol divides the participants of an oracle call into four parties, namely the server, Prover, DECO Verifier, and third-party Verifier. The server represents the off-chain server and stores some information that the on-chain contract wants. Prover represents the user. Prover is responsible for communicating with the server, obtaining data, and then proving the authenticity of the statement about the data to the Verifier. DECO Verifier represents the operator of the DECO oracle and is responsible for verifying the proof provided by the user and signing the statement. The third-party Verifier usually represents the on-chain contract that needs to use off-chain data. The third-party Verifier will receive the signature and statement of the DECO Verifier provided by the Prover, and use the Prover's statement after verifying the signature. The specific details will be described in detail in 3. This capability to bring off-chain data authenticity to the blockchain without compromising user privacy represents a significant advancement towards more sophisticated and privacy-focused DApps.

The DECO protocol is particularly designed to protect users' private data during interactions with off-chain servers, such as API keys and personal information, without requiring modifications to server-side configurations. However, the protocol's proof mechanism inevitably discloses some information to the DECO Verifier. In DECO exist two setting, CBC-HMAC and GCM. For CBC-HMAC, Verifier would learn all the plaintext information except the chunk would be used for the statement. Also, for both setting, Verifier would learn the detail of the statement itself, detail described in 3.3. This will not be a problem if the user trusts DECO, but this disclosure can pose risks in applications requiring high timeliness, such as real-time betting smart contracts. For example, in real-time betting smart contracts which other blockchain user betting on a sport game result, the contract uses DECO to retrieve match result data. If the DECO Verifier identifies the data's source and statement, Verifier could launching a front-run attack. Specifically, Verifier could deliberately delaying signing the statement, gives him time to buying in the wining team on the contract, which gives Verifier unfair competitive advantage. However, even ordinary scenarios for obtaining data can bring privacy risks. For example, if a company wants to use blockchain for large-scale lending, it needs to prove its assets to multiple on-chain lending protocols,

and it will use the oracle to pass statements about its bank balance to the chain multiple times. In this case, DECO Verifier will know that the company is proving its assets multiple times, and then infer the company's financial situation, posing a potential risk of confidentiality leakage.

To enhance the privacy and security of DECO in such scenarios, the goal of this project is to modify the protocol to increase the blindness of the Verifier. This enhancement involves strengthening the privacy features of the Zero-Knowledge Proof (ZKP) component of the protocol to ensure that even in highly time-sensitive applications, the risk of security threats due to information disclosure is mitigated. Such improvements would not only extend the applicability of the DECO protocol but also contribute to advancing blockchain privacy technologies. Optimizing DECO in this manner significantly enhances its suitability and security in specific contexts, laying a more robust foundation for the future application of blockchain and De-Fi technologies.

Chapter 2

Background

2.1 Blockchain & Smart Contract

Blockchain technology initially manifested through Bitcoin [8], representing a decentralized ledger system. As a distributed ledger, the Bitcoin blockchain relies on cryptographic principles and the Proof of Work (PoW) algorithm. Miners solve computational challenges to discover a "nonce" that meets specific criteria, allowing them to generate new blocks containing transaction data. The first miner to successfully resolve the challenge receives newly minted cryptocurrency and transaction fees as rewards. In 2014, Vitalik Buterin introduced the Ethereum white paper [2], which brought the concept of smart contracts—code that executes automatically on the blockchain. Smart contracts have facilitated the rapid growth of decentralized applications (DApps) and decentralized finance (DeFi). DeFi operates as a permissionless decentralized system where all smart contract code is transparent, ensuring transparency and integrity in financial transactions. Notably, blockchains are closed systems, inherently incapable of accessing external networks. When smart contracts require external data, oracles serve as the bridge between the external networks and the blockchain, conveying information to the chain. Also, each operation on the blockchain incurs a gas fee, with more complex codes demanding greater computational power and higher gas costs. Thus, in developing and optimizing the oracle protocol, it is imperative to ensure the accuracy of data and streamline the protocol process as much as possible to reduce computational complexity and costs, thereby enhancing the system's usability.

2.2 Oracle

Most De-Fi systems rely on oracles to obtain external data, such as price feeds, highlighting the significance of oracle security and reliability. The security of many decentralized finance smart contracts depends on the accuracy of the data provided by oracles, such as on-chain identity verification systems and decentralized exchanges. According to research [11], up to 15% of De-Fi attacks from 2018 to 2020 were oracle manipulation. Consequently, research into oracle mechanisms and security has become a focal point in both academic and industrial areas. These studies concern not only the oracles' security aspects but also how they impact the robustness and vulnerabilities of the entire De-Fi ecosystem.

Oracles primarily function as bridges between the external world and the blockchain, providing external data to on-chain De-Fi smart contracts, such as market prices or real-world event information. De-Fi developers can develop their own oracles or use existing public oracle services, such as ChainLink [4]. Oracle operation usually involves off-chain machines collecting internet data and transmitting it to the on-chain oracle. Typically, oracles introduce multiple data providers from different sources to mitigate the impact of malicious actions or errors from individual nodes. Various oracle models have unique operational mechanisms and security measures. The dissertation is focusing on DECO, which will be elaborated on in Chapter 3. These models may involve different trust assumptions, data validation methods, and incentive mechanisms to ensure the reliability of provided data and the effective functioning of on-chain smart contracts.

2.3 Zero Knowledge Proof

Zero-Knowledge Proof is a cryptographic protocol that allows one party, the prover, to prove to another party, the verifier, that a given statement is true without conveying any information apart from the fact that the statement is indeed true. Zero-Knowledge Proofs are characterized by three main properties:

- **Completeness:** If the statement is true, the honest verifier will be convinced by the honest prover.
- **Soundness:** If the statement is false, no cheating prover can convince the honest

verifier that it is true, except with some small probability.

- **Zero-knowledge:** If the statement is true, no verifier learns anything other than the fact that the statement is true. The proof reveals no additional information.

In modern zero-knowledge proofs, the inputs provided to the proof function are typically categorized into public and private inputs. Public inputs are visible in full to the verifier, while private inputs remain undisclosed to the verifier. For example, if a prover wants to demonstrate in zero knowledge that $f(x) = y$ without revealing the value of x , then x would be the private input and y the public input. Formally, this is expressed as:

$$ZK - PoK\{x : f(x) = y\}$$

In this case, the function f is also known to the verifier to facilitate the completion of the verification process. Zero-knowledge proofs are used not only for privacy preservation but also to enable verifiers to quickly verify the correctness of complex function computations. For instance, with STARKs [5], the complexity of generating a proof is $O(n * \text{polylog}(n))$ where n is the number of gates in the circuit, while the verification complexity is $O(\text{polylog}(n))$. It is noteworthy that for verifier efficiency, there are more efficient zero-knowledge proof techniques available, such as Groth'16 [6], which has a verification complexity of only $O(m)$, where m is the length of the public input. However, correspondingly, the proof generation time complexity for Groth'16 is higher than that of STARKs. Thus, when applying zero-knowledge proof algorithms, it is crucial to consider the time complexities for both the prover and verifier.

2.4 Secure Multi-Party Computation

Secure multi-party computation is a cryptographic technology that enables multiple parties to collaboratively compute a function over their private inputs, ensuring that each party's input remains confidential. This cryptographic approach allows participants to achieve the correct output without revealing their individual inputs to one another or to external observers. This paper mainly uses two-party secure computation, in which each participant brings private inputs to the process, and the protocol incorporates public inputs. Public inputs are accessible to both parties, while private inputs are only accessible to the party that provides them. At the end of the computation, both parties obtain respective private outputs and possibly a joint public output. Notably, neither party can only view the private output received by the other, but the value of the

public output is visible to both. In the following chapter, the DECO protocol employs two-party secure computation to facilitate a range of critical functions, including key splitting, TLS packing, and MAC tag computation.

2.5 TLS

TLS (Transport Layer Security) [3] is a widely applied protocol for enhancing the security of internet communications. Its applications span a variety of domains, including web browsing, API calls, and the DECO protocol is relying on the TLS protocol. TLS provides three fundamental functions to ensure communication security: data encryption, data integrity, and authentication. These functions collectively achieve the CIA triad, namely confidentiality, integrity, and availability. Excluding unnecessary contextual details, the TLS communication process can be succinctly described as follows. The version of TLS employed in this study is 1.2, with the key exchange protocol being Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) [1].

1. Key Generation:

The client and server each generate a temporary private key, denoted as s_c and s_s , respectively.

2. Key Exchange:

The key exchange process involves the following steps:

- (a) **ClientHello:** The client sends a “ClientHello” message to the server, containing a client-generated random number r_c .
- (b) **serverHello:** The server responds with a serverHello message, which includes the server’s certificate, a server-generated random number r_s and the server’s public key G and temporary public key Y_s where $Y_s = s_s * G$. Server will also sign on all information.
- (c) **Client’s Temporary Public Key:** Upon receiving the server’s response, the client verify the certificate and signature, replies with its temporary public key Y_c where $Y_c = s_c * G$.

3. Pre-master Secret Calculation:

Both the client and server use the received temporary public key to compute the pre-master secret Z .

- For the server: $Z = s_s * Y_c$
- For the client: $Z = s_c * Y_s$

Due to the properties of elliptic curve operations, both computations yield the same value: $Z = s_c * s_s * G$. Consequently, both parties now share a common pre-master secret.

4. Key Derivation:

The client and server use pre-master secret Z and the previously exchanged random numbers r_c and r_s on pseudo-random function (PRF) to generate the master secret m , and derive the encryption key k^{Enc} and the MAC key k^{MAC} from the master secret m and random numbers r_c and r_s .

5. Completion and Verification:

After computing the encryption and MAC keys, the client and server exchange "ClientFinished" and "serverFinished" messages. These messages are pseudo-random function output using k^{Enc} and include MAC tags generated using k^{MAC} . Upon successful verification of these messages, a secure communication channel is established. The client and server then use the negotiated encryption keys for symmetric encryption of their communication and the MAC keys to generate MAC tags, thereby ensuring the confidentiality and integrity of the transmitted data.

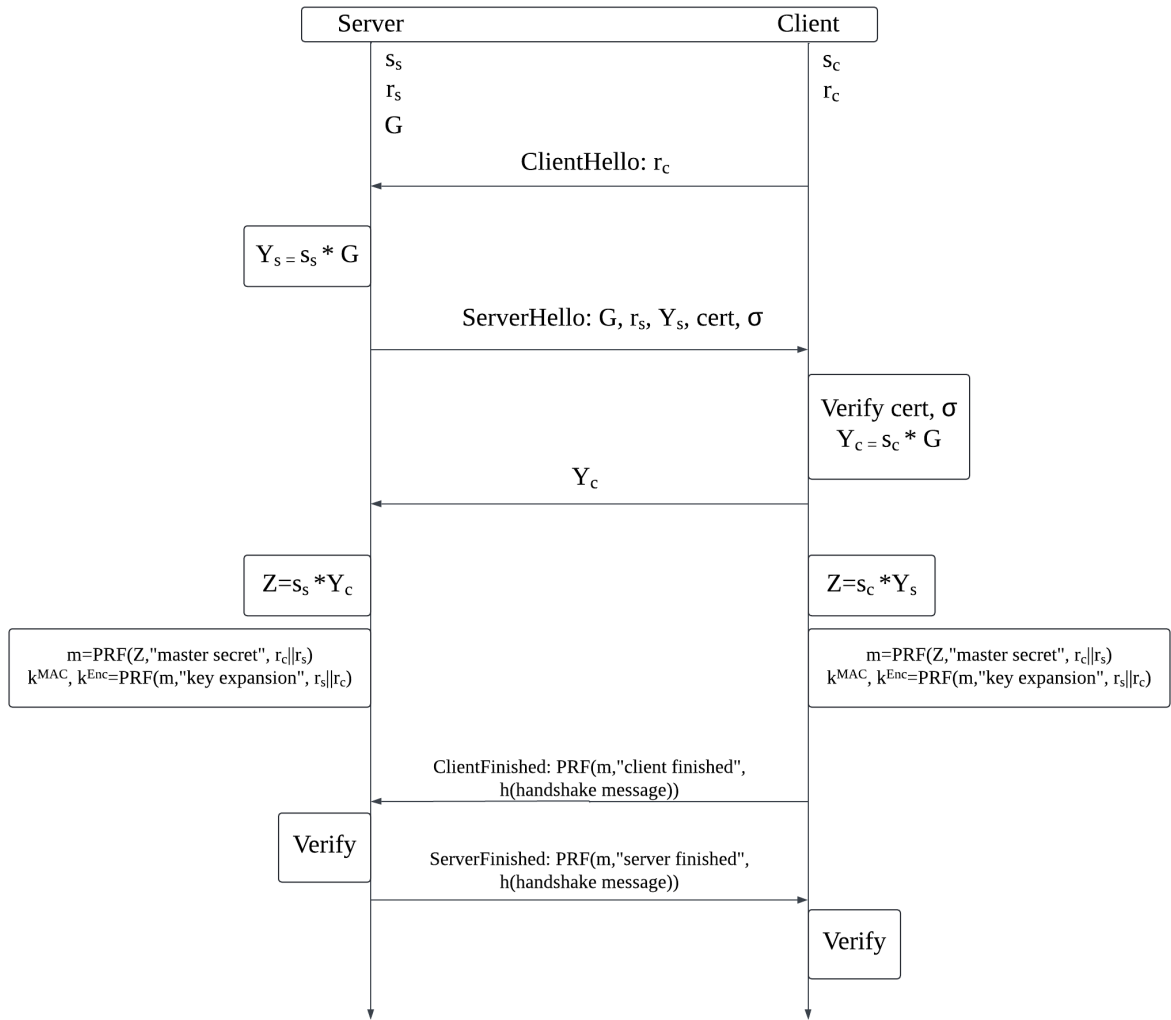


Figure 2.1: TLS Under ECDHE

Chapter 3

DECentralized oracles (DECO)

The DECO protocol is a blockchain oracle that leverages TLS to ensure secure and verifiable data transmission. Its core method involves using cryptographic techniques to partition the TLS key between the Prover and the Verifier, ensuring that neither party can falsify the data from TLS communications without colluding. The Prover is responsible for communicating with the server and proving the correctness of the obtained data to the Verifier. The Verifier, in turn, checks the Prover's proof. Once the communication and proof are complete, the Verifier signs on the statement provided by Prover about the content of the communication. The Prover can then use this verified data to convince others of the authenticity of the data obtained from the server.

The DECO protocol consists of three parts: **Three-Party Handshake**, **Query Execution**, and **Proof Generation**. Details of these three parts are discussed below.

3.1 Three-Party Handshake

The essential idea behind the three-party handshake is to preserve the integrity of the TLS protocol. From the server's perspective, the interaction with the DECO protocol is indistinguishable from a standard TLS process, thereby positioning DECO as a potential oracle protocol for widespread use in blockchain applications. In brief, the process of the DECO three-party handshake is highly similar with TLS handshake, which can be described as follows. Also, the version of TLS is 1.2, with the ECDHE key exchange protocol.

1. Key Generation:

The Prover, Verifier and server each generate a temporary private key, denoted as s_p , s_v and s_s , respectively.

2. Key Exchange:

The key exchange process involves the following steps:

- (a) **ClientHello:** The Prover sends a “ClientHello” message to the server, containing a Prover-generated random number r_c .
- (b) **serverHello:** The server responds with a serverHello message, which includes the server’s certificate, a server-generated random number r_s and the server’s public key G and temporary public key Y_s where $Y_s = s_s * G$. Server will also sign on all information.
- (c) **Prover-Verifier Key Exchange:** The Prover verify the signature and certificate, send random of Prover and server r_s r_c , and server’s public key G and temporary public key Y_s to Verifier. Verifier also verify the signature and certificate, send back its temporary public key: $Y_v = s_v * G$ to Prover.
- (d) **Client’s Temporary Public Key:** Prover compute Prover’s temporary public key $Y_p = s_p * G$, then compute the client’s temporary public key: $Y_c = Y_p + Y_v$, and send to server.

3. Pre-master Secret Calculation:

server use the received temporary public key to compute the pre-master secret Z . And Prover and Verifier compute the share of pre-master secret Z_p and Z_v

- For the server: $Z = s_s * Y_c = s_s * (Y_p + Y_v)$
- For the Prover: $Z_p = s_p * Y_s$
- For the Verifier: $Z_v = s_v * Y_s$

Due to the properties of elliptic curve operations, the computations yield the value.

$$\begin{aligned}
 Z &= s_s * s_p * G + s_s * s_v * G \\
 &= s_s * Y_p + s_s * Y_v \\
 &= s_p * Y_s + s_v * Y_s \\
 &= Z_p + Z_v
 \end{aligned}$$

4. Key Derivation:

For Prover and Verifier, they follow the following steps:

- (a) **Transfer to Field Element:** The goal of this step is to reduce computation cost for subsequent MPC protocol. The addition of Z_p and Z_v in elliptic curve space, they jointly run a protocol ECtF to transfer the addition of Z_p and Z_v in elliptic curve space to cheaper field space. The private input of ECtF is Z_p and Z_v , where $Z = Z_p + Z_v$ in $EC(\mathbb{F}_p)$. ECtF privately output Z_p and Z_v to Prover and Verifier, and output elements addition operation $Z = Z_p + Z_v$ holds in \mathbb{F}_p rather than $EC(\mathbb{F}_p)$ as input.
- (b) **Key Generation and Separation:** Prover and Verifier run a 2-party-computation protocol \mathcal{F}_{2pc}^{hs} . The protocol takes private input Z_p and Z_v from Prover and Verifier, and public input of previously exchanged random numbers r_c and r_s . Protocol use PRF to generate the master secret m , encryption key k^{Enc} and MAC key k^{MAC} from the pre-master secret $Z = Z_p + Z_v$ and random numbers r_c and r_s . Protocol separate the master secret m into m_p and m_v , where $m_p \oplus m_v = m$. Similarly, separate MAC key k^{MAC} into k_p^{MAC} and k_v^{MAC} where $k_p^{MAC} \oplus k_v^{MAC} = k^{MAC}$. Send $(k^{Enc}, k_p^{MAC}, m_p)$ to Prover and (k_v^{MAC}, m_v) to Verifier.

The server simply use pre-master secret Z and previously exchanged random numbers r_c and r_s on PRF to generate master secret m and derive the encryption key k^{Enc} and the MAC key k^{MAC} from the master secret m and r_c and r_s .

5. Completion and Verification:

Similar with original TLS, the server generates a "serverFinished" message, Prover and Verifier run 2-party-computation PRF protocol to generate "ClientFinished" message. Then exchange the finishing message, check the correctness, and abort if message does not match the expectation. Then Prover and Verifier verify the "serverFinished" message through 2-party-computation. By this, three-party handshake complete.

At the end of three-party handshake, Prover holds the complete encryption key and half of MAC key, Verifier holds the other half of MAC key. And at server side, the handshake is no difference with normal TLS handshake, which server gets the complete encryption key and MAC key. By splitting MAC key makes both Prover and Verifier

could not manipulate the content with MAC tag on it, specifically, the response from the server of the query.

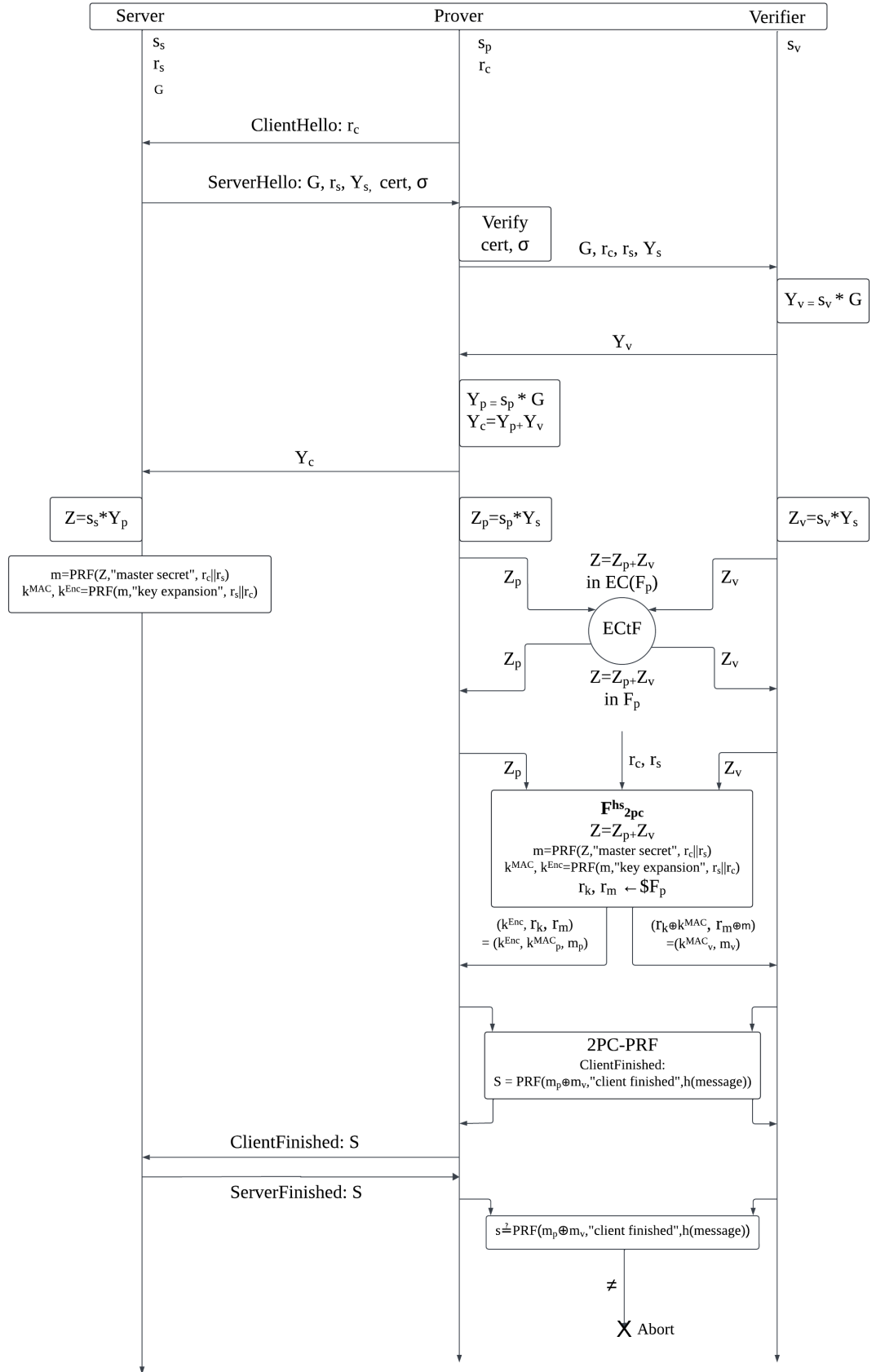


Figure 3.1: Three-Party Handshake Under ECDHE

3.2 Query Execution

Under **CBC-HMAC** setting, every query message need to be sent along with the HMAC tag computed as following.

$$HMAC_H(k, M) = H((k \oplus opad) || H((k \oplus ipad) || M))$$

Where H represents a hash function, and $opad$, $ipad$ represent hard-coded padding parameter. Since MAC key k^{MAC} is shared between Prover and Verifier, a direct approach is simply run 2-party-computation to obtain the HMAC tag. But this would be expensive for large queries. Through the fact of SHA-256 that:

$$H(M_1 || M_2) = H(H(IV, M_1), M_2)$$

A cheaper approach is as following.

1. **Key Hash Computation:** Prover run 2-party-computation with Verifier to obtain the key hash s_0 for Prover where $s_0 = H(IV, k^{MAC} \oplus ipad)$. Since H is one-way hash function, it will not leak k^{MAC} to Prover.
2. **Inner Hash Computation:** Prover compute inner hash by self: $h_i = H(s_0, M)$.
3. **Tag Computation:** Prover and Verifier run 2-party-computation to obtain the outer hash for both party $\tau = H(k^{MAC} \oplus opad || h_i)$, which is the MAC tag.

After obtain the tag τ , Prover uses k^{Enc} to compute and send $(sid, \hat{Q} = Enc(k^{Enc}, M || \tau))$ to server. Server reply with (sid, \hat{R}) . To commit the response to Verifier, they follow the following steps:

1. **Tag Commitment:** Prover send the query and response, along with its share of the MAC key $(sid, \hat{Q}, \hat{R}, k_p^{MAC})$ to Verifier.
2. **Key Recovery:** Verifier send back the other half of MAC key (sid, k_v^{MAC}) . Prover obtain the full MAC key $k^{MAC} = k_v^{MAC} \oplus k_p^{MAC}$.
3. **Tag Verification:** Prover decrypt the respond $R || \tau = Dec(k^{Enc}, \hat{R})$, and verify τ using k^{MAC} .

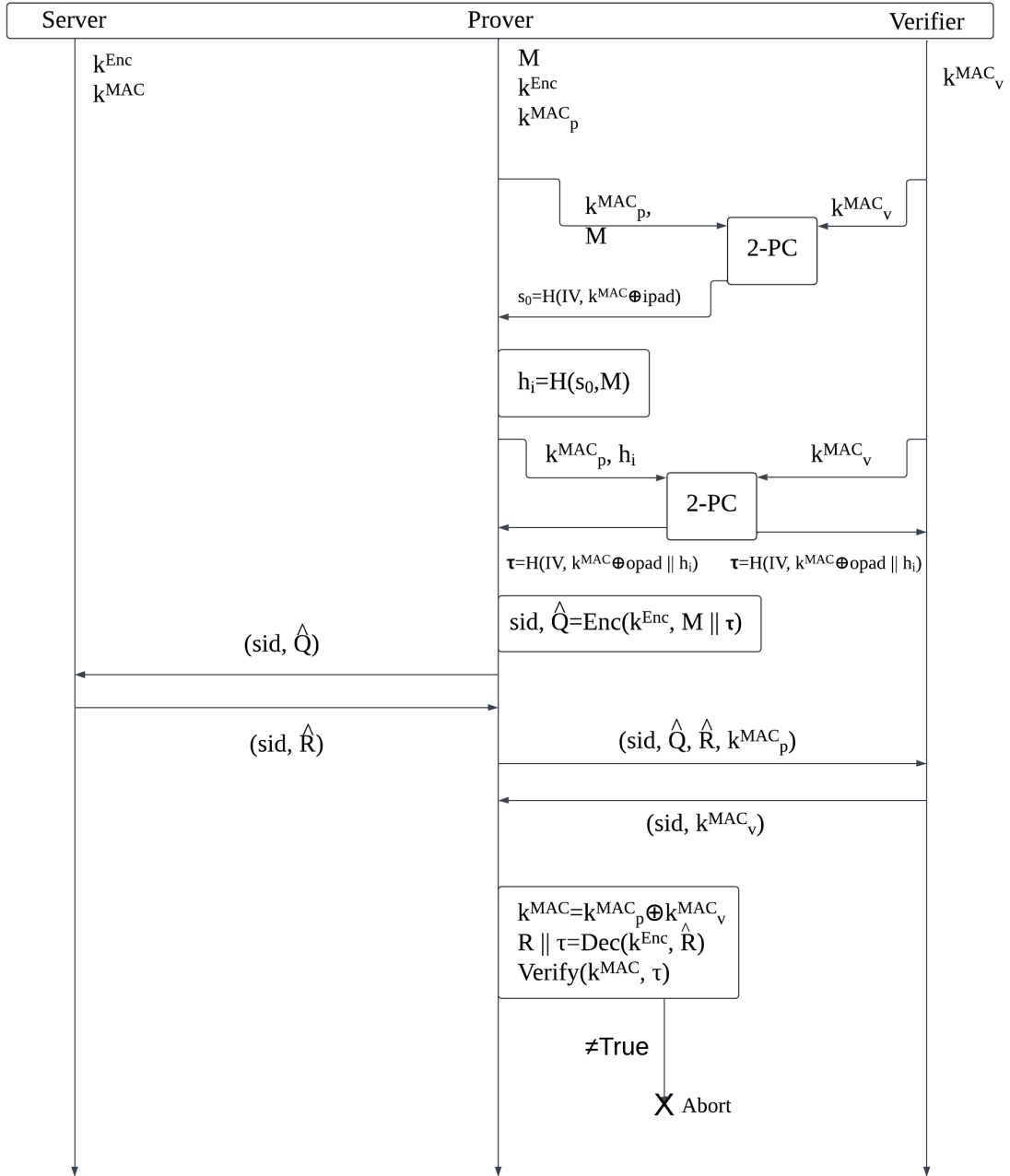


Figure 3.2: Query Execution Under CBC-HMAC

3.3 Proof Generation

After Query Execution, Prover need to prove the statement of the TLS communication he concerned to Verifier without revealing the sensitive information. Verifier will verify the proof, then sign on the statement and send back to Prover. Prover could then use the signed statement to the third party Verifier where the statement will need to be used,

usually an on chain smart contract. Third party Verifier could verify the signiture on the statement, if the signature is from the valid DECO Verifier, Third party Verifier could be convinced the statement is true.

In a scenario where privacy protection is not a concern, after receiving content and a MAC tag from the server, the Prover can directly forward all information along with their share of the MAC key to the Verifier, who can then easily verify the integrity of the information by recompute the MAC tag, since HMAC is collision resistance, Prover could not find another message matches the tag. However, when privacy is considered, the situation becomes significantly more complex. In DECO, the implemetation is as follows:

Under the setting of **CBC-HMAC** with **SHA-256** on TLS, suppose the plaintext after decrypt is $B = \{B_1, B_2, \dots, B_n, \tau\}$, where each B_i represent a 256 bit long block, TLS encryption key is k^{Enc} , MAC key is k^{MAC} and MAC tag is τ , and Prover doesn't want to reveal the content of B_i . Set $B^- = \{B_1, B_2, \dots, B_{i-1}\}$ and $B^+ = \{B_{i+1}, B_{i+2}, \dots, B_n, \tau\}$. Note that at this point, both Prover and Verifier hold the complete MAC key k^{MAC} , and only Prover holding encryption key k^{Enc} . The goal is to generate the zero-knowledge proof of the following:

$$ZK - PoK\{B_i : \tau = HMAC(K^{MAC}, B^- || B_i || B^+)\}$$

Through this proof, Verifier could be convinced block B_i is from the correct TLS communication, then Prover could prove the statement of B_i , without revealing the plaintext information of B_i . To achieve that, DECO does the following.

1. Prover need to prove the following to Verifier to convince Verifier the tag itself is come from the correct ciphertext.

$$\pi_\tau = ZK - PoK\{k_{Enc} : \hat{\tau} = Enc(k_{Enc}, \tau)\}$$

Where $\hat{\tau}$ is the last 3 chunks in ciphertext \hat{M} sent to Verifier before, which is encrypted MAC tag.

2. Prover will also need to prove the statement correctness and chunk where statement is used matches the tag:

$$\pi = ZK - PoK\{B_i : f_{sha256}(h_{i-1}, B_i) = h_i \wedge stmt(B_i)\}$$

Where $h_{i-1} = f_{sha256}(B^-)$, then send $(\pi, h_{i-1}, h_i, B^-, B^+)$ with statement to Verifier.

3. Verifier checks the following content:

- Verifier verify the π and π_τ .
- Verifier recompute the previous content to check if the result $\stackrel{?}{=} h_{i-1}$.
- Verifier recompute the tag using suffix content and MAC key to check if the final result $\stackrel{?}{=} \tau$.

If the verification passes at each step, the Verifier will generate a signature on the statement and send to the Prover or any other party, proof generation process finished.

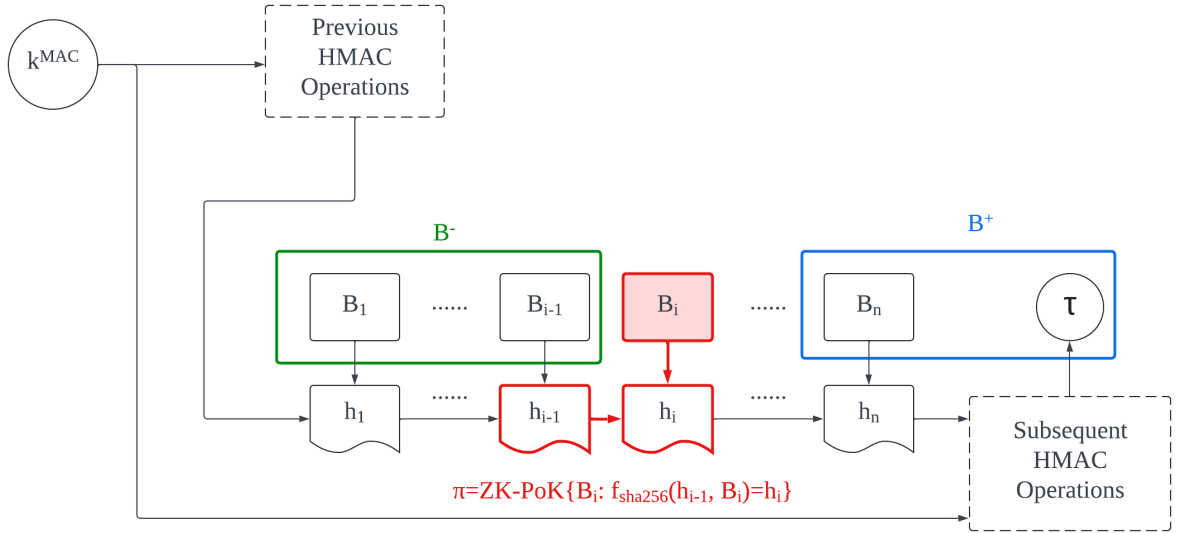


Figure 3.3: Proof Generation Under CBC-HMAC with SHA-256

Chapter 4

Practical Improvement

4.1 Motivation

The DECO protocol is specifically designed to safeguard user data during interactions with off-chain servers, such as API keys and personal details, without requiring changes to server configurations. Although the current implementation effectively achieves this objective, the protocol's proof mechanism inherently discloses certain information to the Verifier.

Specifically, as outlined in the DECO paper, various methods for generating proofs for third parties are described. In the CBC-HMAC setup, if the Prover wishes to conceal sensitive data within a specific SHA-256 chunk, as mentioned in 3, they must reveal all other plaintext data chunks with the statement to the Verifier. Although Appendix A.2 of the DECO paper introduces techniques known as "Redacting a suffix" and "Redacting a prefix", which allow the Prover to hide all data in the prefix or suffix of the sensitive data chunk, this still results in the exposure of multiple data chunks and statement itself in plaintext to the DECO Verifier and Verifier is able to infer a considerable amount of information from the remaining plaintext message. GCM setup of DECO provided a higher lever of privacy, unlike CBC-HMAC, GCM does not have to reveal any data chunk to DECO Verifier. However, the Verifier still learns the content of the proof's statement, posing a notable privacy concern for the Prover.

For instance, as mentioned in the introduction, in smart contracts (third party Verifier) used for betting on sports results, the contract employs DECO to fetch the match results. However, if the DECO Verifier can determine the data's source and intended

use, it might exploit this information. A potential risk arises if the Verifier, knowing the winning team, could delay signing the data verification, allowing them time to place a favorable bet, thereby gaining an unfair competitive advantage. Similarly, in more conventional applications such as large-scale lending, companies might use DECO to demonstrate compliance with certain financial conditions without revealing specific details. Here, the DECO Verifier, becoming aware that the company is frequently proving its assets, might deduce the company's financial situation. Such information, if leaked, could suggest that the company is seeking loans or experiencing financial stress, potentially affecting its stock price and business relationships.

To enhance the privacy and security of DECO in such scenarios, it is recommended to modify the protocol to increase the blindness of the Verifier. This enhancement involves strengthening the privacy features of the Zero-Knowledge Proof (ZKP) component of the protocol, ensuring that even in highly time-sensitive applications, the risk of security threats due to information disclosure is mitigated. Such improvements would not only extend the applicability of the DECO protocol but also contribute to advancing blockchain privacy technologies. Optimizing DECO in this manner significantly enhances its suitability and security in specific contexts, laying a more robust foundation for the application of blockchain and De-Fi technologies in the future.

Research on completely hiding URL addresses has been explored and achieved, with researchers referring to the enhanced version of DECO as PECO [9]. However, the proof could still leak part of the information to Verifier and third party where the data will be used, as mentioned before, either the statement to DECO Verifier, or all other block data. The goal of the project is to completely blind the DECO Verifier, and limit the information third party Verifier get as much as possible, since the data transfer on chain is considered as public. To achieve complete blindness and not leak anything from Verifier, there is another approach, which is not mentioned in the paper or other relevant literature. Stated below.

4.2 Direct Approach

Still under the setting of **CBC-HMAC** with **SHA-256** on TLS 1.2, suppose the reply ciphertext before decrypt is \hat{M} , plaintext message after decrypt is M , and MAC tag is τ . $M || \tau = Dec(k^{Enc}, \hat{M})$. Intuitively, to achieve blindness, Prover and Verifier could

following the following after query execution step:

1. Before the end of query execution, instead of sending encrypted query and respond its share of MAC key $(sid, \hat{Q}, \hat{M}, k_p^{MAC})$ to Verifier, Prover do the AES decryption first, to obtain the plaintext message M with its tag τ from ciphertext $M || \tau = Dec(k^{Enc}, \hat{M})$.
2. Prover send its share of MAC key k_p^{MAC} and MAC tag τ to Verifier
3. Verifier combine the Prover's MAC key with Verifier's MAC key together to obtain the complete MAC key $k^{MAC} = k_p^{MAC} + k_v^{MAC}$, and send k^{MAC} to Prover.
4. Prover prove to the Verifier in zero knowledge of the statement:

$$ZK - PoK\{M; HMAC(K^{MAC}, M) = \tau\}$$

and send the proof to the Verifier with corresponding tag τ .

5. Verifier verifies the proof. If the proof is correct, Verifier put its signature on the tag τ with its own secret key, send the signature $\sigma = sign(\tau)$ back to Prover.
6. After Prover receive the signature σ , to prove the statement to the third party who is concerned, which is usually on-chain, Prove generate the zero knowledge proof of the following:

$$\pi = ZK - PoK\{M; [stmt(M)] \wedge [HMAC(k^{MAC}, M) = \tau]\}$$

(Notice that *stmt* function is a boolean function that proving the content of the message, e.g. if the balance inside the giving message of the account is higher than a certain amount returns True, otherwise False)

7. Prover send the proof π to the third party Verifier, with τ , K^{MAC} and σ . Third party Verifier the proof, and check if the signature is valid.

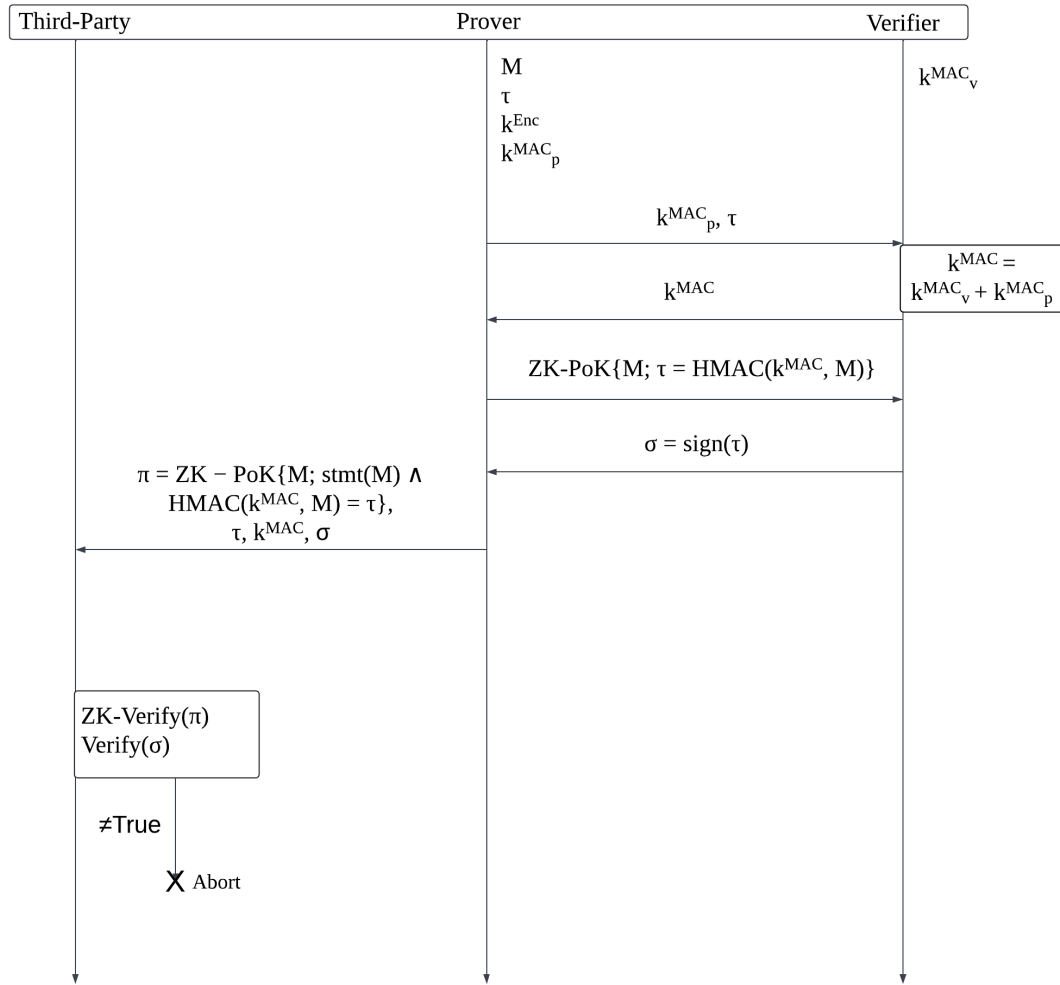


Figure 4.1: Direct Proof on MAC Tag

It can be found that there are two zero knowledge proof involved in this process, also since proof verification time complexity is highly related with the size of the instance or circuit, there exist a way to make optimization, described in 4.3.

4.3 Proof of Integrity on MAC Tag

Apparently, from the perspective of ensuring data integrity, MAC tag is sufficient, and because MAC tag generation is based on SHA-256, with high entropy plaintext, the tag will not convey any information about the plaintext, which is same as 4.2. Therefore, instead of proving the functional relationship between plaintext and tag to Verifier with zero knowledge, we can directly commit the tag to Verifier, and then let Verifier sign the MAC key and tag, which will be sufficient to ensure blindness and security. The

specific protocol is as follows:

1. Before the end of query execution, instead of sending encrypted query and respond and its share of MAC key $(sid, \hat{Q}, \hat{M}, k_p^{MAC})$ to Verifier, Prover do the AES decryption first, to obtain the plaintext message M with its tag τ from ciphertext $M || \tau = Dec(k^{Enc}, \hat{M})$.
2. Prover send its share of MAC key k_p^{MAC} and MAC tag τ to Verifier.
3. Verifier combine the Prover's MAC key with Verifier's MAC key together to obtain the complete MAC key $k^{MAC} = k_p^{MAC} + k_v^{MAC}$, and sign on the MAC key and tag $\sigma = sign(\{k^{MAC}, \tau\})$, send signature to Prover.
4. After Prover receive the signature σ , to prove the statement to the third party who is concerned, which is usually on-chain, Prover generate the zero knowledge proof of the following:

$$\pi = ZK - PoK\{M; [stmt(M)] \wedge [HMAC(k^{MAC}, M) = \tau]\}$$

5. Prover send the proof π to the third party Verifier, with τ and σ . Third Verifier verify the proof, and check if the signature is valid.

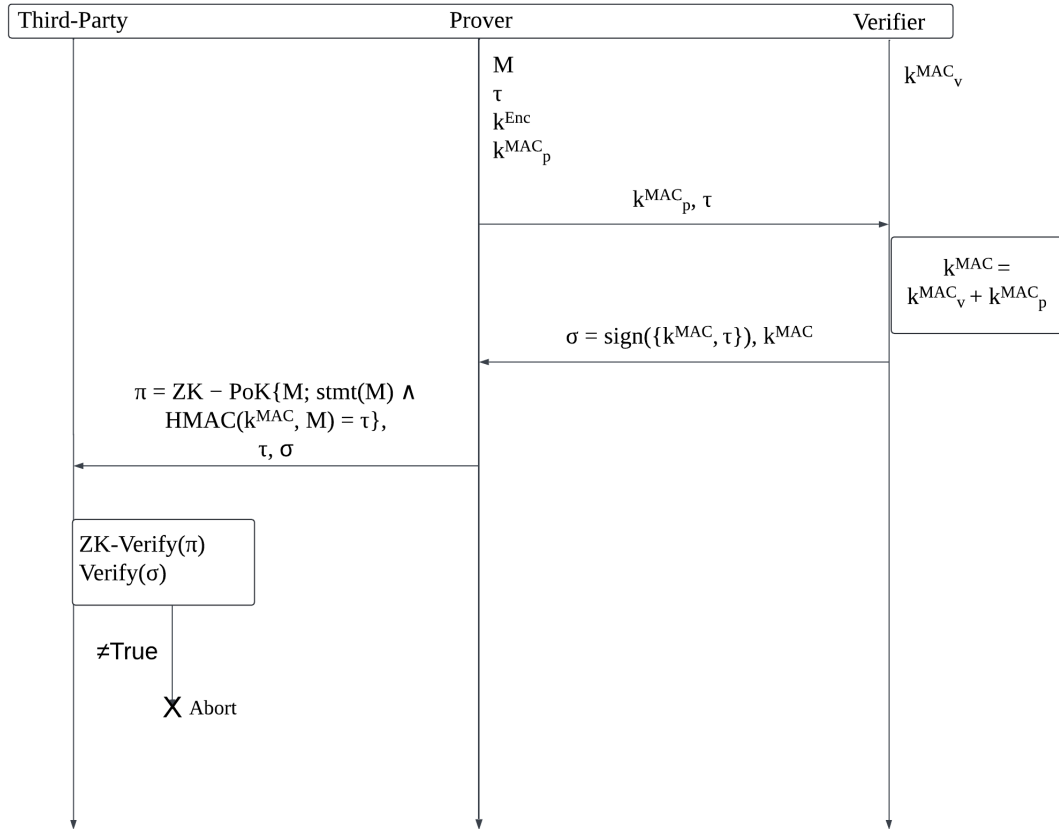


Figure 4.2: Blind Proof on MAC Tag

However, this method brings an obvious problem. This method requires zero-knowledge proof and verification of the SHA-256 of the entire message, and each chunk and its hash in SHA-256 needs to be used as input to form the final hash, which means that if DECO Prover interacts with a website that does not provide an API, the message will be an HTML interface, which may be very large. For the proof task of the Prover, this will not be a problem because the Prover runs off-chain and can handle computationally intensive tasks. But for the verification task of the third-party Verifier, the time complexity of zero-knowledge proof verification is highly related to the instance size. Because the verification will be run on-chain in most cases, the verification of the zero-knowledge proof of the third-party Verifier may be expensive.

Therefore, to deal with this situation, there is another data integrity proof method based on Merkle Root, described in the following section. Using the Merkle root, the Prover can easily prove the contents of a specific chunk to a third-party Verifier while maintaining low proof and verification costs.

4.4 Proof of Integrity on Merkle Trees

Merkle trees represent a method of data digest, utilizing a binary tree structure where the hashes of data blocks are concatenated pairwise, rehashed, and recursively processed until reaching the tree's root, culminating in a fixed-length root hash. This structure offers advantages in terms of efficiency and provability: unlike SHA-256, which requires rehashing all preceding or succeeding blocks to verify the integrity of a particular chunk, Merkle trees enable isolated verification of data block integrity without necessitating a recomputation of the entire sequence of hashes. Using Merkle trees, Verifiers can efficiently validate data block integrity by performing a limited number of hash operations, provided with relevant hashes along the tree path.

In this section, we explore the utilization of Merkle trees to enhance the efficiency and security of the DECO protocol. The DECO Verifier signs on the Merkle root of the plaintext data, thereafter the Prover can utilize this Merkle root to prove specific content about the data without disclosing the full data itself.

1. Before the end of query execution, instead of sending encrypted query and respond and its share of MAC key $(sid, \hat{Q}, \hat{M}, k_p^{MAC})$ to Verifier, Prover do the AES decryption first, to obtain the plaintext message M with its tag τ from ciphertext $M||\tau = Dec(k^{Enc}, \hat{M})$.
2. Prover compute the Merkle root of message $h_{MR} = MH(m)$ send its share of MAC key k_p^{MAC} and MAC tag τ and Merkle root h_{MR} to Verifier.
3. Verifier combine the Prover's MAC key with Verifier's MAC key together to obtain the complete MAC key $k^{MAC} = k_p^{MAC} + k_v^{MAC}$, and send back to Prover.
4. Prover prove to the Verifier in zero knowledge of the statement:

$$ZK - PoK\{M : [HMAC(K^{MAC}, M) = \tau] \wedge [h_{MR} = MH(m)]\}$$

and send the proof to the Verifier.

5. Verifier verifies the proof. If the proof is correct, Verifier put its signature on the Merkle root h_{MR} with its own secret key, send the signature $\sigma = sign(h_{MR})$ back to Prover.

6. Prover can then zero knowledge proof the content in the message with h_{MR} , there are two methods could be use after third party Verifier verifying the signature of DECO Verifier on the Merkle root.

(a) **Direct Proof:** Straightforward way is to locate the chunk where statement needed to be prove, writing as M^i , extract its path to the top root and generate the following proof:

$$\pi = ZK - PoK\{M^i, path; [stmt(M^i)] \wedge [VerifyMerkle(M^i, path, h_{MR}) = True]\}$$

Then send π and path with h_{MR} to Verifier. Detail described in figure 4.3

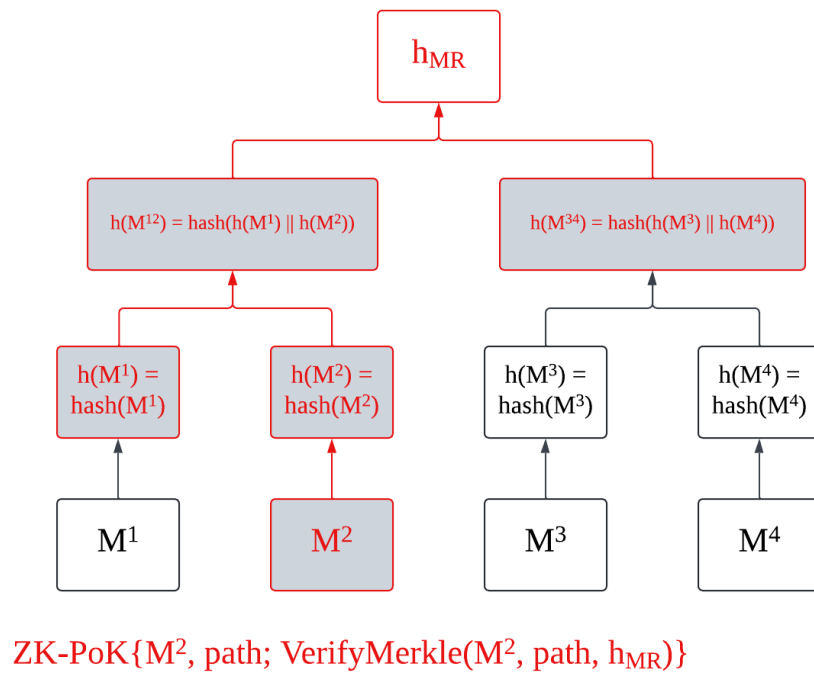


Figure 4.3: Directly Prove on Merkle Tree

(b) **Merkle Root Verify:** This method is by Prover proving the bottom leaf hash operation of the Merkle tree, and send the hash along the path to third party Verifier, Verifier could run Merkle verification function by their own. Verifier would need to verify the ZK proof, and recompute all the hash function along the path by their own. When reaching the root node, the hash is equal to the root DECO verifier signing on, third party Verifier could be convinced the message is correct. Detail described in figure 4.4

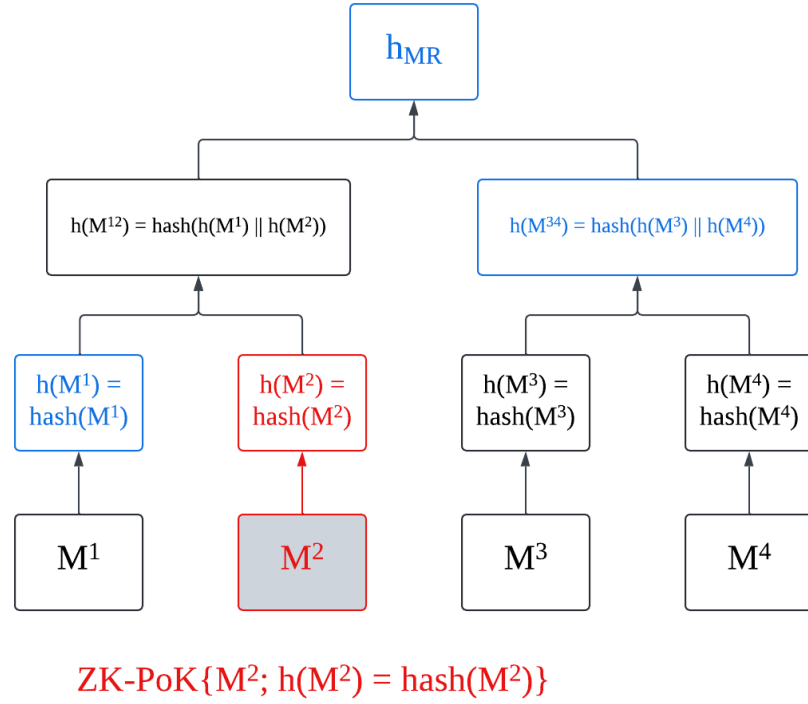


Figure 4.4: Prove Following Merkle Path and Merkle Root
(blue represent the data needed to be sent to Verifier)

Each method has its respective advantages and disadvantages. For method 4.4 6a, the benefit lies in that the third-party Verifier only needs to perform a zero-knowledge proof verification concerning the Merkle path, without requiring additional computations to be assured of the proof's correctness. Utilizing the SNARK method (such as Groth16 [6]), the time complexity is approximately $O(n)$, where n represents the length of the public input. In scenarios involving large Merkle trees, the Prover can opt to treat the entire Merkle path as a private input for the zero-knowledge proof, with the Merkle root as a public input, thus ensuring that the proof verification has a fixed and minimal time complexity. However, the drawback of this method is that it creates a large zero-knowledge proof circuit, imposing significant computational burdens on the Prover. Therefore, this method is more suitable when dealing with longer data lengths and larger Merkle trees. Although the Prover incurs a higher cost in generating the proof, the third-party verifier has minimal costs to bear for verification.

Conversely, method 4.4 6b benefits from the simplicity in proof generation,

involving only the proof of a hash function for a single bottom leaf node, with the Verifier independently verifying the correctness of the Merkle path. This approach relieves the Prover from the computational burden seen in method 4.4 6a. However, this method also has obvious disadvantages; it is not suitable for longer data since the verification process for the Verifier includes calculating the hash for each node along the Merkle path, which needs to be performed $\log l$ times, where l is the length of the data M . Additionally, this method requires the transmission of the values of adjacent nodes along the path, also totaling $\log l$. Although the time and space complexity of $\log l$ is not significantly burdensome, for longer lengths, this cost is non-negligible compared to method 4.4 6a.

In practical applications, considering the total costs for both Prover and Verifier, each method has its appropriate application scenarios. When dealing with long data lengths, method 4.4 6a should be considered to reduce the costs for the third-party Verifier. When the data length is shorter, generating a proof for the entire path does not offer a time cost advantage, thus method 4.4 6b can be used to enhance the speed of proof generation.

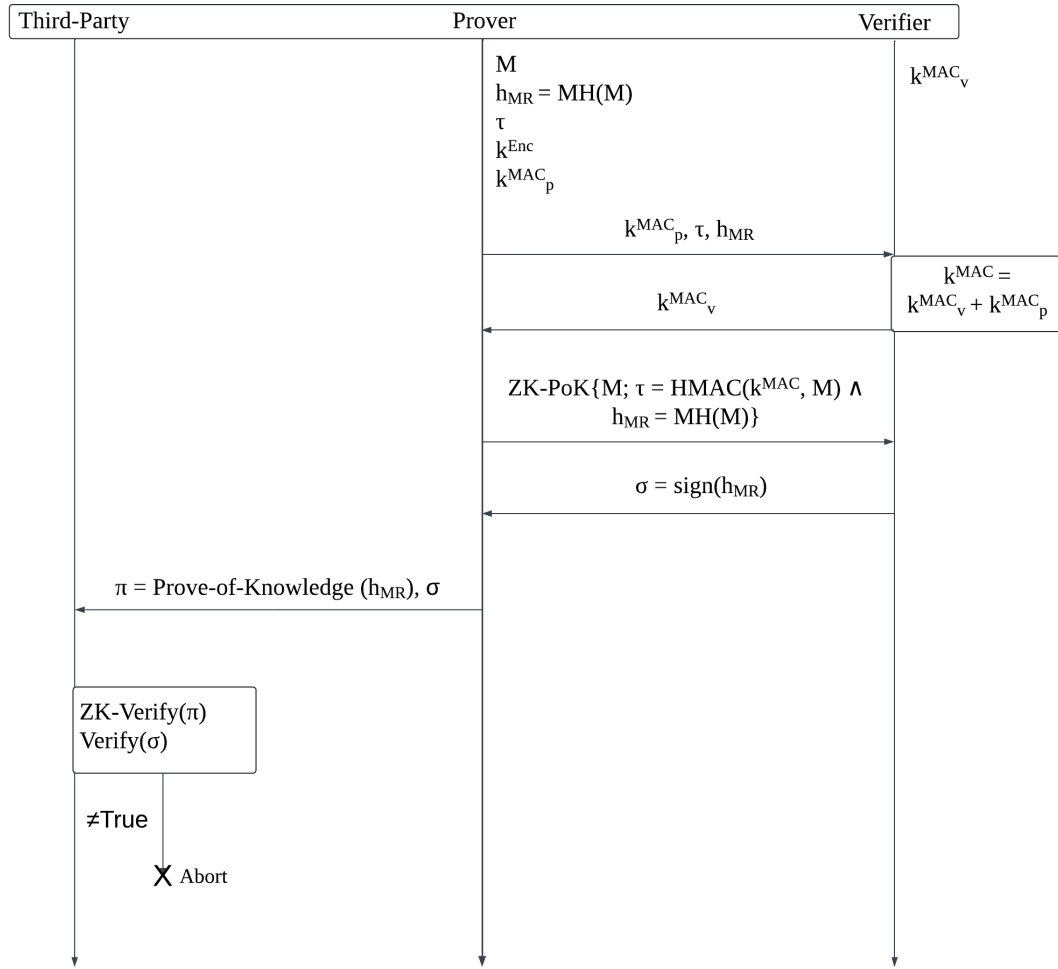


Figure 4.5: Blind Proof of Integrity on Merkle Trees

However, for DECO Verifiers, the scenario is similar to proving on MAC tags, when utilizing HMAC for proof generation, it is imperative that the plaintext data is of high entropy. If not, the Verifier might have the opportunity to deduce the plaintext content through brute-force computation. This requirement adds an additional layer of complexity and potential vulnerability in scenarios where data entropy cannot be guaranteed. This consideration is crucial in practical implementations, as the entropy of data directly influences the security of the system. Insufficient entropy in plaintext, despite the employment of sophisticated encryption or verification techniques, may compromise the security of the system. Therefore when using this method, it is imperative to ensure that all processed data exhibits sufficient randomness and unpredictability to uphold the overall security and reliability of the system.

Another method employs AES-CBC for integrity proofs. Unlike HMAC, the blindness

of AES does not require the high entropy of data but on the secrecy of the encryption and decryption keys. Proofs using AES ciphertext effectively mitigate the issues associated with insufficient data entropy. However, this approach continues to encounter performance difficulties when large data volumes are involved. Moreover, the non-committing property of AES-CBC represents a potential vulnerability. It is essential to prevent the Prover from finding an alternative plaintext-key pair that matches the existing ciphertext. The subsequent section will discuss a proof technique based on AES chunks that is independent of data entropy, with high efficiency for third party Verifiers, and addresses the non-committing issue associated with AES-CBC effectively.

4.5 Proof of Integrity on AES Encryption

It is not feasible to directly obtain a zero-knowledge proof of decryption of AES ciphertext with the relative plaintext, because this requires the encryption process to have committing property. Specifically, committing means that once a ciphertext is generated, it is impossible to find another valid plaintext-key pair to encrypt the same ciphertext without changing the ciphertext, otherwise, Prover could always finding another piece of plaintext that is not belong to the communication with server, but with same ciphertext which Verifier signing on, makes Prover potentially able to prove a statement on forged content. Unfortunately, in the context of this article (CBC-HMAC), symmetric encryption does not have this property. But there is still a way to achieve plaintext commitment and reduce the cost of zero-knowledge proof verification at the same time.

1. At Three-Party Handshake stage, once Prover receive the AES encryption key, Prover need to send a hash of the key $h^{Enc} = Hash(k^{Enc})$ and a random initial vector IV to Verifier to commit the encryption key and IV , so that Prover will not possible finding another valid encryption key and initial vector with same ciphertext.
2. Before the end of query execution, instead of sending encrypted query and respond and its share of MAC key $(sid, \hat{Q}, \hat{M}, k_p^{MAC})$ to Verifier, Prove do the AES decryption first, to obtain the plaintext message M with its tag τ from ciphertext $M || \tau = Dec(k^{Enc}, \hat{M})$.
3. Prover send its share of MAC key k_p^{MAC} , with ciphertext \hat{M} to Verifier. Verifier send back its share of MAC key k_v^{MAC} .

4. In zero knowledge, Prover generate the proof of the following:

$$\begin{aligned}
 & ZK - PoK\{k^{Enc}; [HMAC(k^{MAC}, Dec(k^{Enc}, \hat{M})) = \underbrace{Dec(k^{Enc}, \hat{M})[-1, -2, -3]}_{\text{HMAC tag } \tau}] \\
 & \quad \wedge \\
 & \quad [h^{Enc} = Hash(k^{Enc})]\}
 \end{aligned}
 \tag{4.1}$$

This proof is proving to Verifier that \hat{M} **decrypt with some unknown encrypting session key, the MAC tag of the plaintext under k^{MAC} is the last three chunk of plaintext, and encrypting session key is the one committed before.**

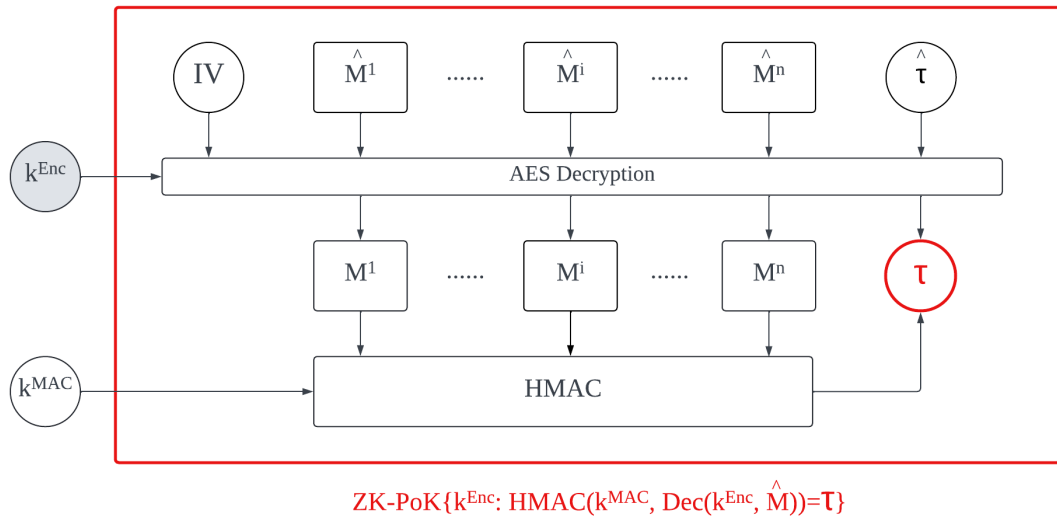


Figure 4.6: First Half of Proof 4.1, Blind Proof of Integrity

5. Prover send the proof to Verifier, also send the chunk index number i where the statement is proved to the third-party Verifier.
6. Verifier verify the proof, then sign on the cipher chunk at index $i - 1$ and i with hash of key h^{Enc} , send signature $\sigma = sign(\{\hat{M}^{i-1}, \hat{M}^i, h^{Enc}\})$ to Prover.
7. Prover generate the proof of the following:

$$ZK - PoK\{k^{Enc}; [Stmt(Dec(k^{Enc}, \hat{M}^i, IV = \hat{M}^{i-1}))] \wedge [h^{Enc} = Hash(k^{Enc})]\}
 \tag{4.2}$$

And send the proof, along with signature σ and hash of key h^{Enc} , ciphertext \hat{M}^i , \hat{M}^{i-1} to third Verifier. Third Verifier verify the proof, and check the signature.

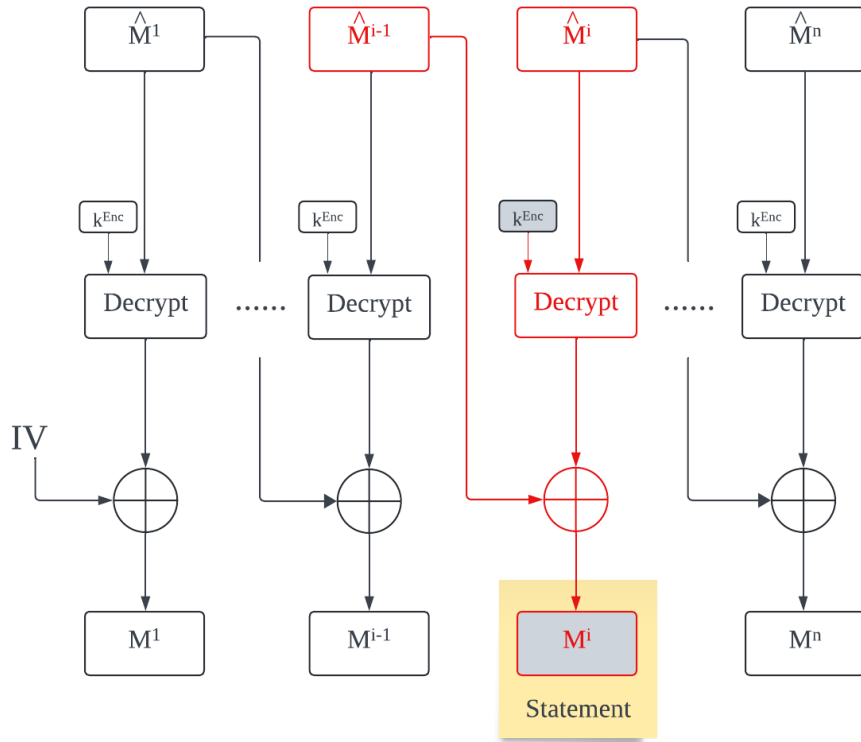


Figure 4.7: First Half of Proof 4.2, Blind Proof of Statement

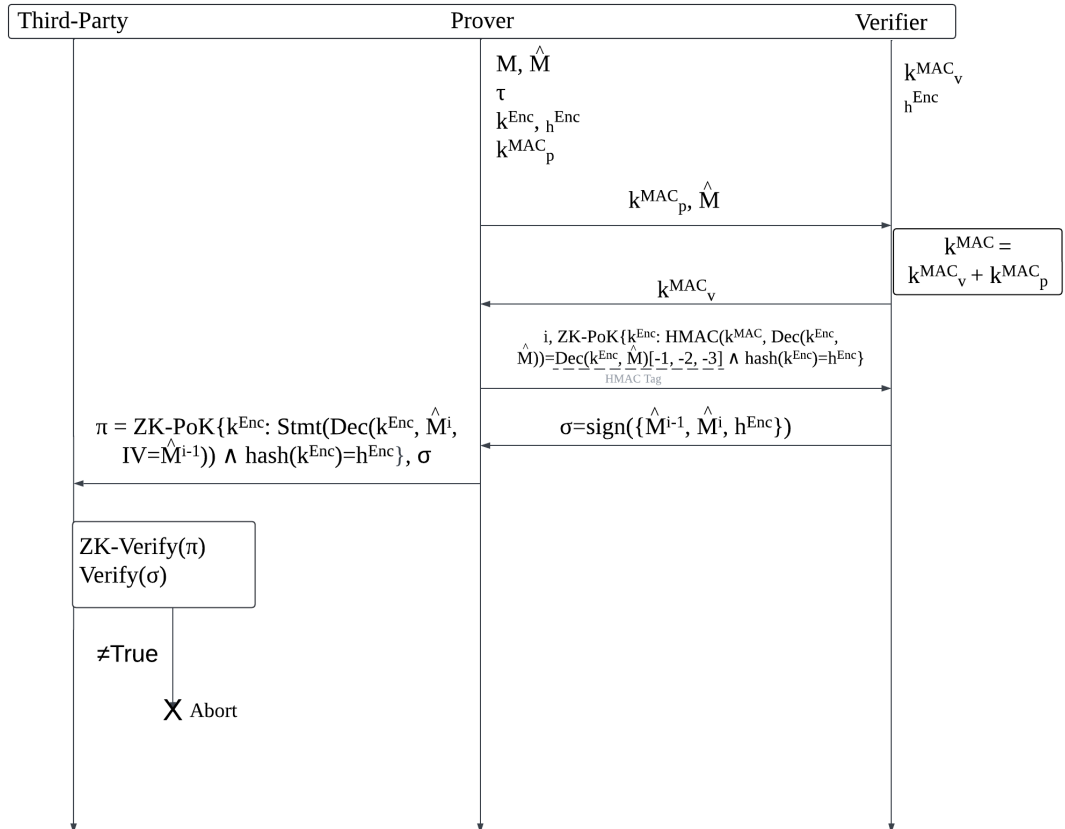


Figure 4.8: Blind Proof of integrity on a Single AES Block

The AES chunk-based proof method has indeed enhanced the security of oracles in environments with low-entropy plaintext, since the blindness is relying on the security of AES key, and has significantly improved the performance of third party Verifiers comparing with Merkle root method and MAC method, since the input size of the ZK proof is only 128 bits constant size long. However, this method has several apparent drawbacks. From a privacy perspective, although the Verifier only knows the specific chunk index within AES-CBC where the required data is located and cannot directly deduce any specific information from the ciphertext, if the Verifier accumulates sufficient historical data and has access to the server, they might use big data analytics to infer the content type stored in the current chunk. For instance, if a bank's website consistently returns data in a fixed format and length, with the balance information always stored in the third AES chunk, the Verifier could deduce that the type of data the Prover is using pertains to specific bank balance information based on the length of the ciphertext and the number of the requested proof block. However, existing a simple way to prevent this. At 4.5 step 5, instead of specifying the block index Prover want to use, Prover could ask Verifier to sign on every single block of AES ciphertext and send to Prover, Prover could use whatever block Prover interested in. Also, by combining the idea of Merkle Root proof with AES proof, same level of privacy can be achieved. This method is mentioned in section 4.7.1.

Moreover, although the verification process might be relatively cheap for third-party third party Verifiers, the proof generation process involved in this method is complex and cumbersome. For both the DECO Prover and Verifier, this process involves the transmission of multiple different key and hash, generation of proofs, and their verification, which not only adds to the operational complexity but also increases the risk of errors, thereby affecting the efficiency and reliability of the entire system. Additionally, since this method entails handling a significant amount computations, involving both MAC computation and AES encryption and decryption, it may require twice resources and time on off-chain machine belongs to Prover and DECO Verifier, thus limiting its widespread application in practical scenarios.

4.6 Comparison

The three methods discussed exhibit variations in performance and privacy, necessitating careful selection based on multiple factors, including the required level of privacy

protection, hardware capabilities, data type, and length. Inappropriate choices may compromise the privacy performance of the protocol or result in unnecessary computational resource expenditure.

Regarding privacy, in high-entropy scenarios, all three protocols achieve blindness, meaning that the Verifier is unable to discern the specific content of the data. However, as previously mentioned, the proof method based on AES reveals the sequence number of the data block of interest. Nonetheless, in low-entropy data scenarios, the methods of proving with a MAC tag and Merkle root face challenges in maintaining data blindness. While protecting privacy in situations of low entropy where sequence numbers cannot be disclosed is challenging, it is not insurmountable; privacy can be maintained by accepting some performance trade-offs through a composite approach, with details to be discussed in the following section.

In terms of hardware performance, the discussion is divided into three parts: the performance of the Prover, the DECO Verifier, and the third party Verifier. The Prover achieves the highest performance using the MAC tag method, as it only requires generating an HMAC proof for the third party Verifier. Both the Merkle tree and AES proof generation are costly since they require providing an HMAC to the DECO Verifier and another prove for third party Verifier (Merkle proof and AES proof) during their respective proof generations. For DECO Verifier, the MAC tag method benefits from the lowest complexity as the DECO Verifier does not require any proof verification, while the DECO Verifier needs to engage in more complex verification processes when dealing with Merkle root and AES methods, both with a complexity of $O(l)$, where l represents the data length. Performance at the third party Verifier level is paramount, as the other two phases are executed off-chain and can handle computationally intensive tasks. On-chain, each computational step incurs gas costs. Here, the MAC tag method has the highest complexity at $O(l)$, as the Verifier needs to validate the SHA-256 hash of the entire data set. Both two Merkle root method, due to its top-down tree structure, has the verification complexity at $O(\log l)$. The AES method, verifying only the process of a single data block, maintains a fixed complexity at $O(1)$. However, it is important to note that we cannot simply choose a method based on overall lower complexity; in scenarios with short data lengths, a simple yet higher complexity method could result in longer overall times due to the extra overhead of generating complex zero-knowledge proofs.

Thus, in summary, the MAC tag method is advisable for proving when data is short and has high entropy due to its better performance. For longer data lengths that also exhibit high entropy, the Merkle tree method is worth considering. When the data is not high in entropy, opting for the more expensive AES method is necessary to ensure data blindness.



















	Prover Performance	DECO Verifier Performance	third party Verifier Performance	High Entropy Dependency	Non Index Leakage	Data Transmitting Size
MAC Tag						
Merkle Root						
AES Block						

Table 4.1: Performance and Characteristics Comparison

4.7 Further Privacy Improvement

4.7.1 Components Composability

The main advantages of these four methods are their flexibility to be combined to meet the requirements of different data types and needs. If we conceptualize the DECO Verifier as a functional module, it can produce various outputs, including signatures for tags, Merkle roots, ciphertexts, and cipher blocks. Depending on different requirements, we can modify the details of each protocol or integrate them with other zero-knowledge proof components. Below is a specific examples.

In extreme scenarios, suppose we want the highest level of privacy, need to process a very long piece of data, the length of which makes direct verification on-chain with a complexity of $O(l)$ impractical. Additionally, this data has low entropy, and Prover does not want the DECO Verifier to know specifically which data block Prover concerned with. In this case, we can combine the Merkle root proof method with the AES block proof method. By initially sending the Merkle root to the DECO Verifier and obtaining

the complete MAC key, we can generate the following proof:

$$\begin{aligned}
 ZK - PoK\{k^{Enc}; [HMAC(k^{MAC}, Dec(k^{Enc}, \hat{M})) = \underbrace{Dec(k^{Enc}, \hat{M})[-1, -2, -3]}_{\text{HMAC tag } \tau}] \\
 \wedge \\
 [h^{Enc} = Hash(k^{Enc})] \\
 \wedge \\
 [h_{MR} = MH(Dec(k^{Enc}, \hat{M}))]\}
 \end{aligned} \tag{4.3}$$

Then Verifier sign on the Merkle root, then the process is same as the Merkle root proof. This combination takes the advantage of blindness of AES cipher with low entropy data, and the advantage of Merkle root proof of none leakage of block index with low verification complexity $O(\log l)$. The trade off is third party Verifier couldn't enjoy the cheapest verification with AES block proof with $O(1)$ complexity.

4.7.2 Unlinkability

Although the above-mentioned method effectively improves the privacy and blindness of DECO, there is still a problem. If the Prover wants to use a DECO Verifier's signature multiple times to prove different statements to the third-party Verifier, the current method will require the Prover to send the same signature to the Verifier multiple times. If the Prover uses different on-chain addresses to complete the operation, the third-party Verifier will be able to link these accounts together and mark them as held by the same party. Similarly, for the DECO Verifier, he can obtain additional information about the user by monitoring on-chain communications and retrieving records of when and where the content he signed was used. This is a disadvantage for users who expect to obtain anonymity in the blockchain. There is a very simple solution to this problem, by using the signature as a private input and integrating the verification process into the zero-knowledge proof process provided to the third party. For example, for the proof provided to the third-party Verifier in 4.3 with signature $\sigma = sign(\{k^{MAC}, \tau\})$.

$$\pi = ZK - PoK\{M; [stmt(M)] \wedge [HMAC(k^{MAC}, M) = \tau]\}$$

Proof can be change to the following:

$$\pi = ZK - PoK\{M, \sigma; [stmt(M)] \wedge [HMAC(k^{MAC}, M) = \tau] \wedge [verify(pk_{DECO}, \sigma)]\}$$

Where pk_{DECO} is the public key of DECO Verifier, which is the public input to the proof. Through this improvement, third party Verifier and DECO Verifier could only

know the content is signed by one of the DECO Verifier, but could not link any two proof with the same signature together, providing DECO with a higher level of blindness and privacy.

Chapter 5

Security

The security of this protocol includes **Completeness**, **Soundness**, and **Blindness**. In this protocol, in the context of this article, Completeness means that if the statement to be proved by the Prover is true, an honest prover can always convince the honest Verifier. Soundness means that if the statement to be proved by the Prover is false, no fraudulent prover can convince the honest Verifier, except for a very small probability. And Blindness means that neither the DECO Verifier nor the third-party Verifier can obtain any information other than the statement from their communication.

5.1 Completeness

5.2 Soundness

5.3 Blindness

Chapter 6

Limitation & Conclusion

Bibliography

- [1] Joppe W Bos, J Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18*, pages 157–175. Springer, 2014.
- [2] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.
- [3] Tim Dierks and Eric Rescorla. The transport layer security (tls) protocol version 1.2. Technical report, 2008.
- [4] Steve Ellis, Ari Juels, and Sergey Nazarov. Chainlink: A decentralized oracle network. *Retrieved March*, 11:2018, 2017.
- [5] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo—a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, 2021.
- [6] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology—EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 305–326. Springer, 2016.
- [7] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1348–1366. IEEE, 2021.
- [8] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

- [9] Manuel B Santos. Peco: methods to enhance the privacy of deco protocol. *Cryptography ePrint Archive*, 2022.
- [10] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. Deco: Liberating web data using decentralized oracles for tls. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1919–1938, 2020.
- [11] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2444–2461. IEEE, 2023.