

DECentralized oracles (DECO): Extensions and Applications

Wenxing Duan



Master of Science
School of Informatics
University of Edinburgh
2024

Abstract

This paper focuses on a blockchain oracle protocol, Decentralized Oracles (DECO). It provides a detailed analysis of DECO's operational mechanisms, highlighting its deficiencies and potential privacy risks. Addressing these concerns, the paper proposes four enhancement strategies and meticulously explains the operational principles and processes of these methods. The advantages and disadvantages of each method are extensively compared and analyzed, and their effectiveness is demonstrated through code validation. Additionally, the paper defines the security attributes of the enhanced protocol and includes a simulator-based proof of its blindness properties in the appendices.

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	6
2.4	Secure Multi-Party Computation	7
2.5	TLS	7
3	DECentralized oracles (DECO)	9
3.1	Three-Party Handshake	9
3.2	Query Execution	12
3.3	Proof Generation	13
4	Practical Improvement	15
4.1	Motivation	15
4.2	Direct Approach	16
4.3	Proof of Integrity on MAC Tag	18
4.4	Proof of Integrity on Merkle Trees	19
4.5	Proof of Integrity on AES Encryption	24
4.6	Code implementation	27
4.7	Comparison	28
4.8	Further Privacy Improvement	29
4.8.1	Components Composability	29
4.8.2	Unlinkability	30
5	Security	32
5.1	Completeness	32

5.2	Soundness	32
5.2.1	Soundness for the Prover	33
5.2.2	Soundness for the DECO Verifier	35
5.3	Blindness	35
5.3.1	Blindness for the DECO Verifier	36
5.3.2	Blindness for Third Party Verifier	38
6	Conclusion & Limitation	39
	Bibliography	41
A	Protocol Diagram	43
A.1	TLS	44
A.2	DECO Three-Party Handshake	45
A.3	DECO Query Execution	46
A.4	Direct Approach	47
A.5	Proof of Integrity on MAC Tag	48
A.6	Proof of Integrity on Merkle Trees	49
A.7	Proof of Integrity on AES Encryption	50
B	Blindness Security Proof	51
B.1	Blindness for the DECO Verifier	51
B.1.1	Direct Approach	52
B.1.2	MAC Tag Approach	53
B.1.3	Merkle Tree Approach	53
B.1.4	AES Approach	54
B.2	Blindness for the third party Verifier	56
B.2.1	Direct Approach	56
B.2.2	MAC Tag Approach	57
B.2.3	Merkle Tree Approach	57
B.2.4	AES Approach	58

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 [11]. By 2014, Ethereum [3] 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) [3]. 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. However, because of the trustless nature of blockchain limits smart contracts from accessing external internet data, smart contracts builder cannot simply upload the off chain data into the contract by their own, since the user of the contract would questioning the authenticity of data. This key feature of blockchain necessitating the use of oracles for data provisioning. 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 [10]. Consequently, enhancing the speed, cost-efficiency, privacy, and accuracy of oracle services represents a critical research priority in the blockchain domain. This project specifically focuses on improving the privacy aspects of a recently developed oracle protocol known as DECO (DECentralized Oracle) [13].

DECO is 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 a proof generated through a three-party TLS handshake protocol, involving a proof Verifier using smart contract, DECO enables smart contracts to authenticate data without the willing involvement of data providers. Specifically, the DECO protocol divides the participants of an oracle call into four parties, namely the Server, the Prover, the DECO Verifier, and the third party Verifier. The Server represents the off-chain web 2.0 server and stores some information that the on-chain contract wants. The the Prover represents the user. The the Prover is responsible for communicating with the Server, obtaining data, and then proving the authenticity of the statement about the data to the third party Verifier. The 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 the third party Verifier usually represents the on-chain contract that needs to use off-chain data. The 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. But in practice, the idea of DECO could be expanded for intranet or other kind of system, where the third party Verifier would represent the place where data been verified used and processed. The specific details will be described in detail in chapter 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 has been specifically designed to protect users' private data (such as API keys and personal information) during interactions with off-chain Servers, without requiring modifications to Server-side configurations. However, the current protocol's proof mechanism discloses some information to the DECO Verifier. DECO support two TLS setting, CBC-HMAC and GCM. For CBC-HMAC, the DECO Verifier would learn all the plaintext information except the chunk that would be used for the statement. Also, for both setting, the DECO Verifier would learn the detail of the statement itself, as described in more detail in section 3.3. This will not be a problem if the user trusts DECO, but this disclosure can pose risks, when the DECO Verifier having conflicting

interest, e.g, in applications requiring high timeliness, such as decentralized trading contracts. Consider a blockchain-based contract for trading a gold token, where the token price is strictly pegged to the real-world price of gold, and the contract utilizes DECO to retrieve gold price data. If the DECO Verifier identifies the data's source and statement, the DECO Verifier or other party could launch a front-run attack. Specifically, upon detecting that the decentralized exchange is requesting gold prices and observing significant price increases in the incoming data, the DECO Verifier could deliberately delay signing the statement to give himself time to buy the gold token with lower price. This gives the DECO Verifier an unfair competitive advantage. This attack also exists in other oracle systems [7], and DECO has not been able to completely solve this problem. However, even ordinary scenarios for obtaining data can bring privacy risks. As another example, if a company wants to use a 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, the 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 leaking business critical confidential information.

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 DECO 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

The DECO protocol is designed for blockchain oracles, primarily to provide proof of data origin to smart contracts (third party Verifiers). Through DECO, users (the Prover) can prove that data originates from specific website addresses via TLS. Moreover, DECO enables users to prove statements about the data to others without disclosing the data itself. Most importantly, the DECO protocol does not require any modifications to the TLS protocol by data providers (servers), meaning it is applicable to any website that supports TLS. This chapter will introduce the background knowledge necessary to understand the DECO protocol.

2.1 Blockchain & Smart Contract

Bitcoin [11], created in 2009 by Satoshi Nakamoto, is a decentralized digital currency designed to function as a peer-to-peer electronic cash system. Blockchain technology, serving as the foundation of Bitcoin, is a distributed ledger technology that ensures the transparency and security of all transactions. 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 [3], which introduced the concept of smart contracts—code that executes automatically on the blockchain. Smart contracts enable the 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. However, the

blockchain itself does not have the ability to access the Internet. In order to obtain off-chain data, it is necessary to manually upload the data. However, due to their trustless nature, when smart contracts require external data, data cannot be uploaded directly by smart contract developers because this requires users to trust the developer's data integrity. In this case, oracles serve as a trusted third party act 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 [14], 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 [6]. 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 [8], 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 [9], 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) [4] 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, and diagram can be found at **Appendix A.1**. The version of TLS employed in this study is 1.2, with the key exchange protocol being Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) [2].

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.

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 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 the Prover about the content of the communication. The 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 and diagram can be found at **Appendix A.2**. Also, the version of TLS is 1.2, with the ECDHE key exchange protocol.

1. Key Generation:

The 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 the Prover sends a “ClientHello” message to the Server, containing a the 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) **the Prover-Verifier Key Exchange:** The the Prover verify the signature and certificate, send random of the 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 the Prover.
- (d) **Client’s Temporary Public Key:** the Prover compute the 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 the 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 the 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 the 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:** the Prover and Verifier run a 2-party-computation protocol \mathcal{F}_{2pc}^{hs} . The protocol takes private input Z_p and Z_v from the 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 the 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, the 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 the Prover and Verifier verify the "ServerFinished" message through 2-party-computation. By this, three-party handshake complete.

At the end of three-party handshake, the 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 the Prover and Verifier

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

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 the 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, where diagram can be found at **Appendix A.3**.

1. **Key Hash Computation:** the Prover run 2-party-computation with Verifier to obtain the key hash s_0 for the Prover where $s_0 = H(IV, k^{MAC} \oplus ipad)$. Since H is one-way hash function, it will not leak k^{MAC} to the Prover.
2. **Inner Hash Computation:** the Prover compute inner hash by self: $h_i = H(s_0, M)$.
3. **Tag Computation:** the 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 , the 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:** the 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}) . The Prover obtain the full MAC key $k^{MAC} = k_v^{MAC} \oplus k_p^{MAC}$.
3. **Tag Verification:** the Prover decrypt the respond $R || \tau = Dec(k^{Enc}, \hat{R})$, and verify τ using k^{MAC} .

3.3 Proof Generation

After Query Execution, the 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 the Prover. The 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. The third party Verifier could verify the signiture on the statement, if the signature is from the valid the DECO Verifier, the 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, the 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 the 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 the Prover and Verifier hold the complete MAC key k^{MAC} , and only the 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 the Prover could prove the statement of B_i , without revealing the plaintext information of B_i . To achieve that, DECO does the following.

1. the 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.

- the 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.

- 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.

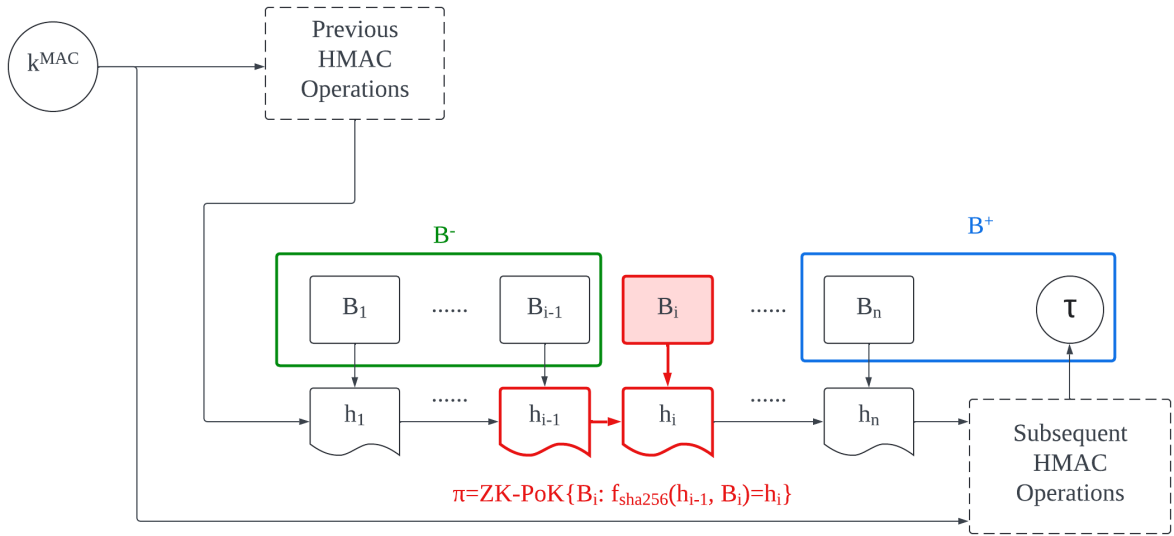


Figure 3.1: Proof Generation Under CBC-HMAC with SHA-256

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 DECO 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 DECO 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 the DECO 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 the DECO Verifier. However, the DECO 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 (the third party Verifier) used for gold token trading, 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 DECO Verifier, knowing the increase of gold price, could delay signing the data verification, allowing them time to buy in the gold token, 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 DECO 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 [12]. However, the proof could still leak part of the information to the DECO Verifier and third party where the data will be used, as mentioned before, either the statement to the DECO Verifier, or all other block data. The goal of the project is to completely blind the DECO Verifier, and limit the information the third party Verifier gets as much as possible, since the data transfer on chain is considered as public. To achieve complete blindness and not leak anything from the DECO 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, the Prover and the DECO Verifier and the third party Verifier could following the following after query execution step, where diagram can be found at **Appendix A.4**:

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 the DECO Verifier, the 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. the Prover send its share of MAC key k_p^{MAC} and MAC tag τ to the DECO Verifier
3. the DECO Verifier combine the Prover's MAC key with the DECO Verifier's MAC key together to obtain the complete MAC key $k^{MAC} = k_p^{MAC} + k_v^{MAC}$, and send k^{MAC} to the Prover.
4. the Prover prove to the DECO Verifier in zero-knowledge of the statement:

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

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

5. the DECO Verifier verifies the proof. If the proof is correct, the DECO Verifier put its signature on the tag τ with its own secret key, send the signature $\sigma = sign(\tau)$ back to the Prover.
6. After the 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. the Prover send the proof π to the third party Verifier, with τ , k^{MAC} and σ . The third party Verifier the proof, and check if the signature is valid.

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 the DECO Verifier with zero-knowledge, we can directly commit the tag to the DECO Verifier, and then let the DECO Verifier sign the MAC key and tag, which will be sufficient to ensure blindness and security. The specific protocol is as follows, , where diagram can be found at **Appendix A.5**:

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 the DECO Verifier, the 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. the Prover send its share of MAC key k_p^{MAC} and MAC tag τ to the DECO Verifier.
3. the DECO Verifier combine the Prover's MAC key with the DECO 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 the Prover.
4. After the Prover receive the signature σ , to prove the statement to the third party who is concerned, which is usually on-chain, the Prover generate the zero-knowledge proof of the following:

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

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

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 the 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 the 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. Diagram can be found at **Appendix A.6**

In this section, we explore the utilization of Merkle trees to enhance the efficiency and security of the DECO protocol. The 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 the DECO Verifier, Prove do the AES decryption first, to obtain the plaintext message M with its tag τ from ciphertext $M||\tau = Dec(k^{Enc}, \hat{M})$.
2. the 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 the DECO Verifier.
3. the DECO Verifier combine the Prover's MAC key with the DECO Verifier's

MAC key together to obtain the complete MAC key $k^{MAC} = k_p^{MAC} + k_v^{MAC}$, and send back to the Prover.

4. the Prover prove to the DECO 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 DECO Verifier.

5. the DECO Verifier verifies the proof. If the proof is correct, the DECO Verifier put its signature on the Merkle root h_{MR} with its own secret key, send the signature $\sigma = \text{sign}(h_{MR})$ back to the Prover.
6. the Prover can then zero-knowledge proof the content in the message with h_{MR} , there are two methods could be use after the third party Verifier verifying the signature of the DECO Verifier on the Merkle root.

- (a) **Prove of Complete Path:** 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 the third party Verifier. Detail described in figure 4.1

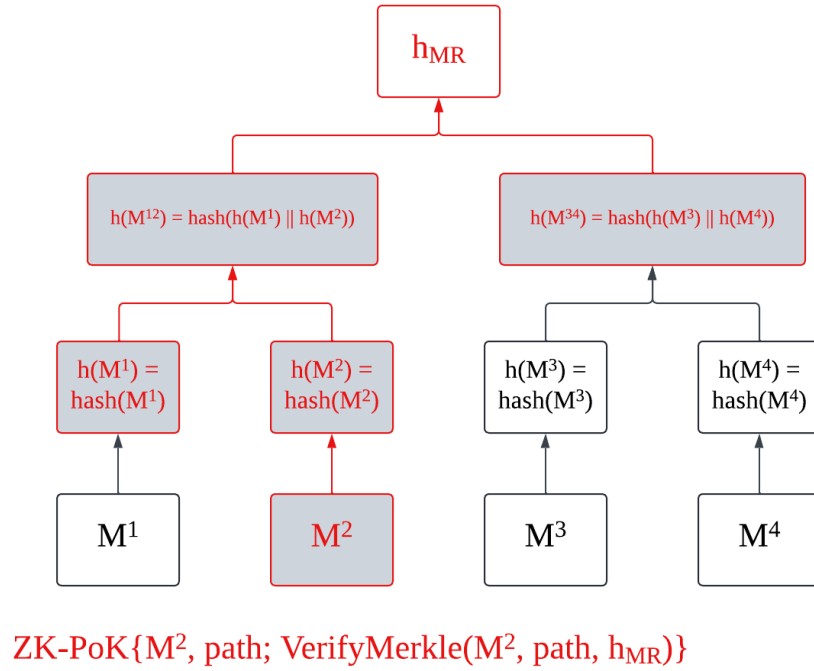


Figure 4.1: Directly Prove on Merkle Tree

- (b) **Prove of Leaf Node:** This method is by the Prover proving the bottom leaf hash operation of the Merkle tree, and send the hash along the path to the third party Verifier, the third party Verifier could run Merkle verification function by their own. The third party 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, the third party Verifier could be convinced the message is correct. Detail described in figure 4.2

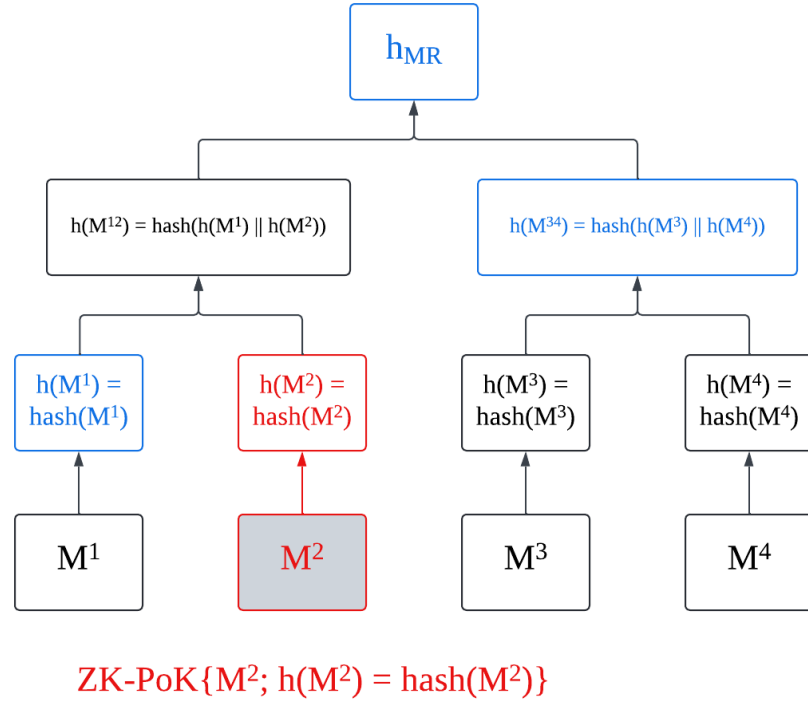


Figure 4.2: Prove Following Merkle Path and Merkle Root
(blue represent the data needed to be sent to the third party Verifier)

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 [9]), 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 third party 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 third party 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 the Prover and the third party 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.

However, for the 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 the 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, the Prover could always finding another piece of plaintext that is not belong to the communication with Server, but with same ciphertext which the DECO Verifier signing on, makes the 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. Diagram can be found at **Appendix A.7**.

1. At Three-Party Handshake stage, once the Prover receive the AES encryption key, the Prover need to send a hash of the key $h^{Enc} = Hash(k^{Enc})$ and a random initial vector IV to the DECO Verifier to commit the encryption key and IV , so that the 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 the DECO 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. the Prover send its share of MAC key k_p^{MAC} , with ciphertext \hat{M} to the DECO Verifier. The DECO Verifier send back its share of MAC key k_v^{MAC} .

4. In zero-knowledge, the 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 the DECO 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.**

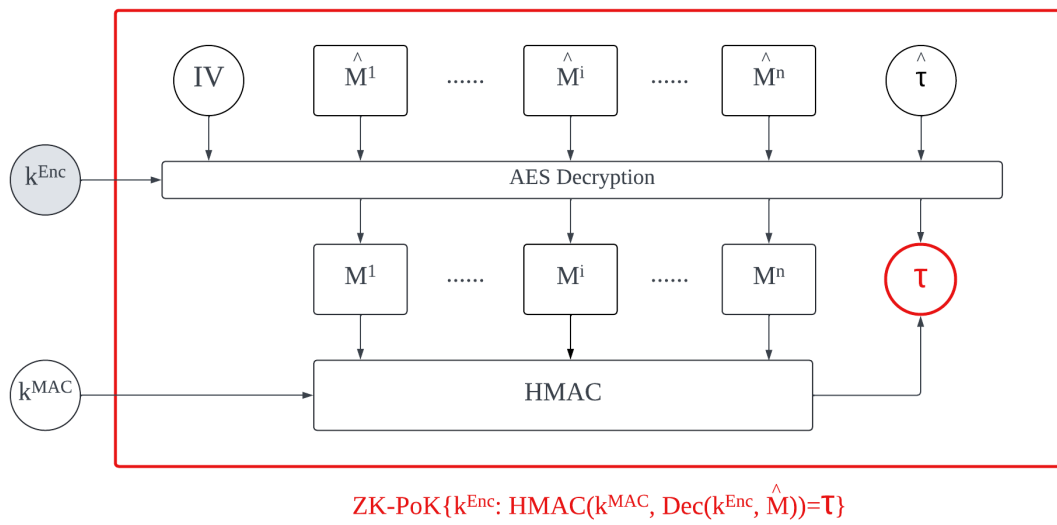
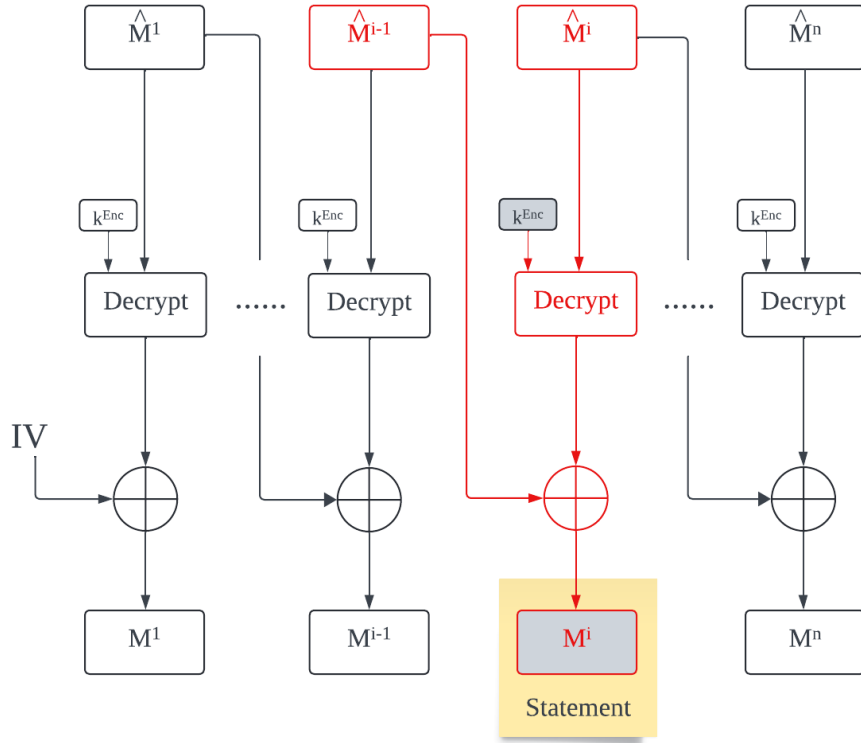


Figure 4.3: First Half of Proof 4.1, Blind Proof of Integrity

5. the Prover send the proof to the DECO Verifier, also send the chunk index number i where the statement is proved to the third party Verifier.
6. the DECO 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 the Prover.
7. the Prover generate the proof of the following:

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

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



$$\text{ZK-PoK}\{k^{Enc}; \text{Stmt}(\text{Dec}(k^{Enc}, \hat{M}^i, \text{IV}=\hat{M}^{i-1}))\}$$

Figure 4.4: First Half of Proof 4.2, Blind Proof of Statement

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 the 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 DECO 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 DECO 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 DECO 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 the Prover want to use, the Prover could ask the DECO Verifier to sign on every single block of AES ciphertext and send to the Prover, the Prover could use whatever block the 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 4.8.1.

Moreover, although the verification process might be relatively cheap for third party the third party Verifiers, the proof generation process involved in this method is complex and cumbersome. For both the DECO the Prover and the third party 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 the Prover and the DECO Verifier, thus limiting its widespread application in practical scenarios.

4.6 Code implementation

To verify the feasibility and compare the performance of the proposed methods, this paper employed two different proof systems to code-validate the four methods: Cairo [8], based on STARKs [1], and Zokrate [5], using Groth16 [9]. STARKs primarily benefit from eliminating the need for a trusted setup but at the cost of higher proof and verification time complexities. In contrast, Zokrate based on Groth16 offers lower time complexities for both proof generation and verification, albeit requiring a trusted setup. In practical applications, Cairo is utilized on StarkNet, whereas Zokrate is employed on Ethereum. On a test machine equipped with a 12900k CPU and 64GB of memory, protocols written in Cairo typically require 10-20 seconds to generate proofs and 0.1 seconds for verification; Zokrate, however, needs only 0.5 seconds

for proof generation, with verification times also under 0.1 seconds. Through code implementation, this project has validated the feasibility of the concept, demonstrating that the four methods can enhance the privacy of the DECO protocol while ensuring reliability. It is important to note that although the protocol code was primarily written to prove concept feasibility and has not reached the optimization levels of actual business and production environments, the brief execution times further demonstrate the practical viability of these methods in real-world applications.

4.7 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 DECO 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 the third party Verifier (Merkle proof and AES proof) during their respective proof generations. For the 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 third party 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.



















	the Prover Performance	the DECO Verifier Performance	the 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

4.8 Further Privacy Improvement

4.8.1 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 the Prover does not want the DECO Verifier to know specifically which data block the 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 the DECO 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 the third party Verifier couldn't enjoy the cheapest verification with AES block proof with $O(1)$ complexity.

4.8.2 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 the 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 $\sigma = \text{sign}(\{k^{MAC}, \tau\})$.

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

Proof can be change to the following:

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

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

A secure blind DECO protocol should satisfy three properties: **Completeness**, **Soundness**, and **Blindness**. 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 with a very small probability. Finally, Blindness means that neither the DECO Verifier nor the third party Verifier can obtain any information other than the output from the protocol (MAC key, MAC tag, Merkle root, ciphertext and/or hash of encryption key for the DECO Verifier in different methods; statement about the data, MAC tag, MAC key, ciphertext and/or hash of encryption key for the third party Verifier).

5.1 Completeness

if the Prover, the DECO Verifier and the third party Verifier following the protocol, for the DECO Verifier, Completeness requires that, for the protocol, the probability of the DECO Verifier accepting the proof provided by the Prover and agree to signing on the corresponding content is close to 1. And for the third party Verifier, completeness represent that the probability of the third party Verifier accepting the proof and signature signed by the DECO Verifier provided by the Prover is close to 1.

5.2 Soundness

Discussing the soundness for a corrupted Server and the third party Verifier does not have much practical significance, since Server could always generate the forged TLS

record with correct encryption key and MAC tag, and the third party Verifier is the party requesting for the data through the Prover. Therefore, the Server and the third party Verifier would pre-set to be honest here. The discussion of soundness is divided into two aspects, soundness for the Prover and soundness for the DECO Verifier.

5.2.1 Soundness for the Prover

In the improved scheme, the soundness for the Prover means that during communication, if the zero-knowledge proof provided by the Prover to the DECO Verifier and the content for which the Prover seeks the DECO Verifier's signature do not belong to the current TLS communication with the Server, then the probability that the DECO Verifier accepts this proof and signs the content is negligible. Consequently, the probability that a third party Verifier can validate the zero-knowledge proof and the signature is also negligible. The following will respectively elaborate on the four schemes mentioned in chapter 4, the Prover is playing the role of adversary, trying to prove on the message that is not belong to the current TLS session with Server, possibly with different MAC key, MAC tag and encryption key. The following will show why it is not feasible.

1. For the direct approach mentioned in 4.2, soundness could be described as following: For the message M , MAC key k^{MAC} and MAC tag τ , if any one of these three is not matching with the correct one, the probability of both Verifier accepting the proof is negligible. The MAC key is one of the public input to the proof, and the Prover only holding half of the MAC key before sending it to the DECO Verifier and get the reply of other half. Therefore, the MAC key holding by the DECO Verifier can not be tampered by the Prover. Even when the Prover sending its share of MAC key, the Prover decided to send a fake share, the Prover still doesn't know the key computed by the DECO Verifier by combining the fake share with Verifier's share that is unknown to the Prover, which leads to the failure of proof verification. For message and tag, since the tag is sent to the DECO Verifier along with its share of MAC key at the same time, by the property of MAC, the Prover cannot pre-generate the valid fake tag with tampered message without knowing the complete MAC key, and after knowing the key, the tag has been committed to the DECO Verifier, and finding another message resulting with the same tag meaning a collision, which has negligible probability.
2. For the MAC tag proof approach mentioned in 4.3, soundness could be described as following: For the message M , MAC key k^{MAC} and MAC tag τ , if any one of

these three is not matching with the correct one, the probability of both Verifier accepting the proof is negligible. One key difference is in this approach, the DECO Verifier didn't involve in the proof verification. The DECO Verifier simply signing on the MAC tag and key computed using the share sent by the Prover. For the same reason mentioned before, the Prover cannot tamper the MAC key, or generate a valid MAC tag without knowing the complete MAC key. Also the Prover cannot find a collision of MAC function, which grants the integrity of the message, key and tag.

3. For the Merkle root proof approach mentioned in 4.4, soundness could be described as following: For the message M , MAC key k^{MAC} Merkle root h_{MR} and MAC tag τ , if any one of these three is not matching with the correct one, the probability of both Verifier accepting the proof is negligible. Same as before, the Prover need to sent its share of MAC key, tag and Merkle root to the DECO Verifier before getting the complete MAC key, which makes the key, tag and message unforgeable, since it cannot finding a collision of the MAC tag. Also in the proof π_p sent to the DECO Verifier, the Prover also need to prove the message that matching with the tag also matching with the Merkle root. If the Prover able to use another Merkle root and prove its matching relation with the tag and message, meaning the Prover either break the zero-knowledge proof protocol, or finding a collision on Merkle root, which is negligible probability.
4. For the AES proof approach mentioned in 4.5, soundness could be described as following: For the message M , MAC key k^{MAC} and encryption key k^{Enc} , if any one of these three is not matching with the correct one, the probability of both Verifier accepting the proof is negligible. Before the Prover sending its share of MAC key, the Prover need to commit the hash of the encryption key to the DECO Verifier. But instead of sending the MAC tag to Verifier, the Prover send the ciphertext along with its share of MAC key. Since the Prover did not hold the complete MAC key, the Prover cannot forge a valid tag and put into the ciphertext, nor finding another message matches the tag and put into the ciphertext. Also, since the encryption key has been committed before through hash, the Prover cannot finding another valid encryption key to decrypt the ciphertext to get another different message and matches the hash sent before at the same time. Also, the content inside the ciphertext has to match with the MAC tag contained inside, since symmetric encryption is bijection function with fixed encryption key, only

one valid ciphertext exist, which fixed the ciphertext. Therefore, the ciphertext and encryption (decryption) key has been fixed, and the Prover will not be able to generate the statement on another fake message.

5.2.2 Soundness for the DECO Verifier

The soundness for the DECO Verifier requires if the DECO Verifier acting maliciously and signing on the content that is not belonging to the current TLS session, the probability the signature been accepted by the Prover and third Party Verifier is negligible. The signature is signing on the content that computed (Merkle root in 4.4), decrypted (MAC tag in 4.2 and 4.3) or forwarded (AES block in 4.5) by the Prover, which will be described in the following.

For the Merkle root, the Merkle root is computed from the decryption of the ciphertext receiving from Server, which cannot be tampered by the DECO Verifier. The decryption key is computed from secure 2-party computation protocol \mathcal{F}_{2pc}^{hs} , to tamper the decryption key, the DECO Verifier would need to break \mathcal{F}_{2pc}^{hs} , which by the assumption of DECO, is negligible probability. Therefore the ciphertext and decryption key are fixed, Merkle root of the plaintext message decrypted from ciphertext using decryption key is also fixed, which gives the DECO Verifier no chance of manipulation.

For MAC tag, since the tag is decrypted from ciphertext. As mentioned in the previous paragraph, ciphertext and decryption key cannot be tampered by the DECO Verifier, which make it also unable to tamper the MAC tag, which is the information inside ciphertext.

For AES block, the ciphertext is directed received from the Server by the Prover, if the DECO Verifier trying to signing on another different AES ciphertext, it will be noticed by the Prover immediately, which make the DECO Verifier unable to tamper and sign the AES block.

5.3 Blindness

In this thesis, the blindness for two party need to be considered, the blindness for the DECO Verifier and blindness for the third party Verifier. For the DECO Verifier,

blindness indicate that during the communication with the Prover, with polynomial time, the DECO Verifier could not learn anything about the information about the message the Prover is using. For the third party Verifier, blindness requires that during the communication with the Prover, with polynomial time, the only information the third party Verifier can learn about the message is the statement. Details of the blindness for two Verifiers is described in the following section, and proof can be found at **Appendix B**.

5.3.1 Blindness for the DECO Verifier

Blindness for the DECO Verifier is defined as the following. Given two communication records on improved DECO scheme, the message of one of the record is sampled from random. The probability of the DECO Verifier distinguish the one with actual message is close to 50%.

1. For the direct approach mentioned in 4.2, blindness could be described as following:

$$\pi_p^0 = ZK - PoK\{r; HMAC(k^{MAC^0}, r) = \tau^0\}, r \leftarrow \{0, 1\}$$

$$\pi_p^1 = ZK - PoK\{M; HMAC(k^{MAC^1}, M) = \tau^1\}$$

$$R^0 = \{\pi^0, k^{MAC^0}, \tau^0\}, R^1 = \{\pi^1, k^{MAC^1}, \tau^1\}$$

$$Pr[V(R^0) = 1] \approx Pr[V(R^1) = 1]$$

Function V is a distinguishing function output the records index, R^0 and R^1 are two communication records that is visible to the DECO Verifier (All data the DECO Verifier can see during the session) on direct approach method. Records with index 0 is the one contained random message. As mentioned in 4.2, this method requires high entropy of message to ensure the blindness. By the property of HMAC function, if message is high entropy, the HMAC tag of the message is indistinguishable with the tag of random. Also by the Zero-Knowledge property of zero-knowledge proof protocol, proof will not leak the information of the private input, which is the message. Therefore two zero-knowledge proof π_p^0 and π_p^1 are indistinguishable, thus gives two records indistinguishability. Consequently, if the random message record π_p^0 is indistinguishable with record π_p^1 that contain message, the DECO Verifier wouldn't able to learn any information about the message using the data he received (DECO records).

2. For the MAC tag proof approach mentioned in 4.3, blindness could be described as following:

$$HMAC(k^{MAC^0}, r) = \tau^0, HMAC(k^{MAC^1}, M) = \tau^1, r \leftarrow \{0, 1\}$$

$$R^0 = \{k^{MAC^0}, \tau^0\}, R^1 = \{k^{MAC^1}, \tau^1\}$$

$$Pr[V(R^0) = 1] \approx Pr[V(R^1) = 1]$$

The data that is visible to Verifier contains only the MAC key and MAC tag, same as mentioned in the last paragraph, if the message is high entropy, two MAC tag are indistinguishable, which leads to the blindness for the DECO Verifier.

3. For the Merkle root proof approach mentioned in 4.4, blindness could be described as following:

$$\pi_p^0 = ZK - PoK\{r : [HMAC(k^{MAC^0}, r) = \tau^0] \wedge [h_{MR}^0 = MH(r)]\}, r \leftarrow \{0, 1\}$$

$$\pi_p^1 = ZK - PoK\{M : [HMAC(k^{MAC^1}, M) = \tau^1] \wedge [h_{MR}^1 = MH(M)]\}$$

$$R^0 = \{\pi^0, k^{MAC^0}, \tau^0, h_{MR}^0\}, R^1 = \{\pi^1, k^{MAC^1}, \tau^1, h_{MR}^1\}$$

$$Pr[V(R^0) = 1] \approx Pr[V(R^1) = 1]$$

Same as before, proofs and tags are indistinguishable. For Merkle root h_{MR}^0 and h_{MR}^1 , the Merkle root of the message with high entropy would be also indistinguishable with the Merkle root of random. Therefore, the record of message is indistinguishable with the record with the random, which indicates the blindness for the DECO Verifier.

4. For the AES proof approach mentioned in 4.5, blindness could be described as following:

$$\pi_p^0 = ZK - PoK\{k^{Enc^0}; [HMAC(k^{MAC^0}, Dec(k^{Enc^0}, \hat{M}^0)) = \underbrace{Dec(k^{Enc^0}, \hat{M}^0)[-1, -2, -3]}_{HMAC \text{ tag } \tau^0}] \wedge [h^{Enc^0} = Hash(k^{Enc^0})]\}$$

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

$$\hat{M}^0 = Enc(k^{Enc^0}, r || HMAC(k^{MAC^0}, r)), r \leftarrow \{0, 1\}$$

$$\hat{M}^1 = Enc(k^{Enc^1}, M || HMAC(k^{MAC^1}, M))$$

$$R^0 = \{\pi^0, k^{MAC^0}, h^{Enc^0}, \hat{M}^0\}, R^1 = \{\pi^1, k^{MAC^1}, h^{Enc^1}, \hat{M}^1\}$$

$$Pr[V(R^0) = 1] \approx Pr[V(R^1) = 1]$$

Same as before, two zero-knowledge proofs are indistinguishable. For the ciphertext \hat{M}^0 and \hat{M}^1 , since CBC-AES is CPA secure, when encryption is kept secret, two pieces of ciphertext are indistinguishable. Also, since the encryption key is generated using PRF (mentioned in chapter 3), extracting the key from their hash would also be unfeasible, which gives this method blindness guarantee.

5.3.2 Blindness for Third Party Verifier

On the side of the third party Verifier, the information transmitted is highly analogous to that received by the DECO Verifier, with the primary distinction being the specific zero-knowledge proofs conveyed. For both the direct approach and the MAC Tag method, the transmitted information to the third party Verifier includes a zero-knowledge proof π , which encompasses private inputs: the message M , MAC tag τ , MAC key k^{MAC} , and the signature of the DECO Verifier on the MAC key and tag, paralleling the description in 5.3.1. This ensures that no additional information about the message can be deduced from the data received by the third party Verifier, other than the validated statement itself.

In the Merkle root method, besides the standard data received by the DECO Verifier (MAC key, MAC tag, Merkle root), the third party Verifier also gains access to the Merkle tree path, which is a sequence of hashes. Given the high entropy of the message, disclosing the Merkle path does not compromise the protocol's blindness.

For the AES method, the information that the third party Verifier receives is identical to that received by the DECO Verifier, except for the proof which discloses the statement. The substantial similarity in the information received by both verifiers suggests that the level of security in terms of blindness provided by the DECO Verifier is equivalently extended to the third party Verifier, with the exception of statement disclosure.

Thus, from a security standpoint, there is no significant distinction in terms of blindness between the two verifiers, only the third party Verifier would be able to learn the statement.

Chapter 6

Conclusion & Limitation

In conclusion, this thesis focuses on the decentralized oracle DECO, primarily examining its privacy aspects. A detailed analysis of DECO's operational principles has been conducted, identifying several privacy concerns inherent in its framework. Specifically, DECO's protocol necessitates that the Prover inevitably reveals certain information to the Verifier during the proof provision process. For the CBC-HMAC setup, the Verifier becomes aware of the statement that the Prover needs to prove to a third party and a portion of the plaintext message. In the GCM setup, while there is no plaintext leakage, the Verifier still becomes aware of the Prover's statement. In certain scenarios, such privacy breaches could have severe consequences, ranging from the leakage of corporate confidential information to creating substantial arbitrage opportunities for malicious Verifiers.

To address these issues, the thesis proposes four solutions that effectively enhance DECO's privacy capabilities, each with its advantages and disadvantages. The simplest is the direct approach, which employs zero-knowledge proof to demonstrate the correspondence between the MAC tag and plaintext, assuring the DECO Verifier of the legitimacy of the provided tag, accompanied by the DECO Verifier's signature. The MAC tag method simplifies this process by omitting the proof to the DECO Verifier. As the Prover does not hold the complete MAC key when the MAC tag is sent to the DECO Verifier, it is impossible for the Prover to forge a valid MAC tag at that moment. The DECO Verifier only needs to ensure that it does not send its portion of the MAC key to the Prover before signing the tag and MAC key. With a signed tag and MAC key, the Prover can then generate a proof for the third party Verifier, including the statement about the message and its correspondence with the MAC key and tag. Although this method is straightforward, its time complexity is high, and the proof generation time

can be burdensome for the Prover. To mitigate this, the thesis proposes a third solution based on Merkle path proof. The the Prover proves the correspondence between the plaintext and MAC tag to the DECO Verifier while also calculating the Merkle root of the plaintext message and proving its correctness to the DECO Verifier. After the DECO Verifier signs the Merkle root, the Prover can generate a proof for the third party Verifier via the Merkle path. This method reduces the data length input into the zero-knowledge proof protocol from $O(n)$ to $O(\log n)$, significantly enhancing operational efficiency. However, these three methods' privacy relies on high entropy of the message, which could pose risks of brute force attacks if the entropy is insufficient. The AES method addresses this by having the Prover commit an encryption key to the Verifier beforehand and then proving the correspondence between the ciphertext and plaintext. Since the tag is also encrypted along with the message into the ciphertext, the Prover can demonstrate the correspondence between the message and tag in the plaintext without revealing the tag or message. Subsequently, the Verifier signs the specified block of ciphertext, and the Prover uses this block to decrypt and prove the statement to the third party Verifier. This method does not require high entropy in the data and also simplifies the proof process. Although this method discloses the sequence number of the ciphertext block of the specific statement, the paper also proposes a solution to this issue by requiring the DECO Verifier to sign all ciphertexts.

Despite these four methods indeed enhancing DECO's privacy and blindness, some limitations still exist. Firstly, these methods lack public verifiability, still requiring the third party Verifier to trust the DECO Verifier. Additionally, if the Prover and the DECO Verifier collude, they could generate false proofs and signatures undetected. To mitigate this possibility, it would be necessary to increase the number of participants in a single session of the DECO Verifiers. However, this approach requires multiple communications when multiple the DECO Verifiers are involved, rather than a unified multi-party protocol that could perform all functions collectively, a direction that still requires further exploration.

Bibliography

- [1] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.
- [2] 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.
- [3] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.
- [4] Tim Dierks and Eric Rescorla. The transport layer security (tls) protocol version 1.2. Technical report, 2008.
- [5] Jacob Eberhardt and Stefan Tai. Zokrates-scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091. IEEE, 2018.
- [6] Steve Ellis, Ari Juels, and Sergey Nazarov. Chainlink: A decentralized oracle network. *Retrieved March*, 11:2018, 2017.
- [7] Shayan Eskandari, Mehdi Salehi, Wanyun Catherine Gu, and Jeremy Clark. Sok: Oracles from the ground truth to market manipulation. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 127–141, 2021.
- [8] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo—a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, 2021.

- [9] 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.
- [10] 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.
- [11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.
- [12] Manuel B Santos. Peco: methods to enhance the privacy of deco protocol. *Cryptology ePrint Archive*, 2022.
- [13] 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.
- [14] 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.

Appendix A

Protocol Diagram

A.1 TLS

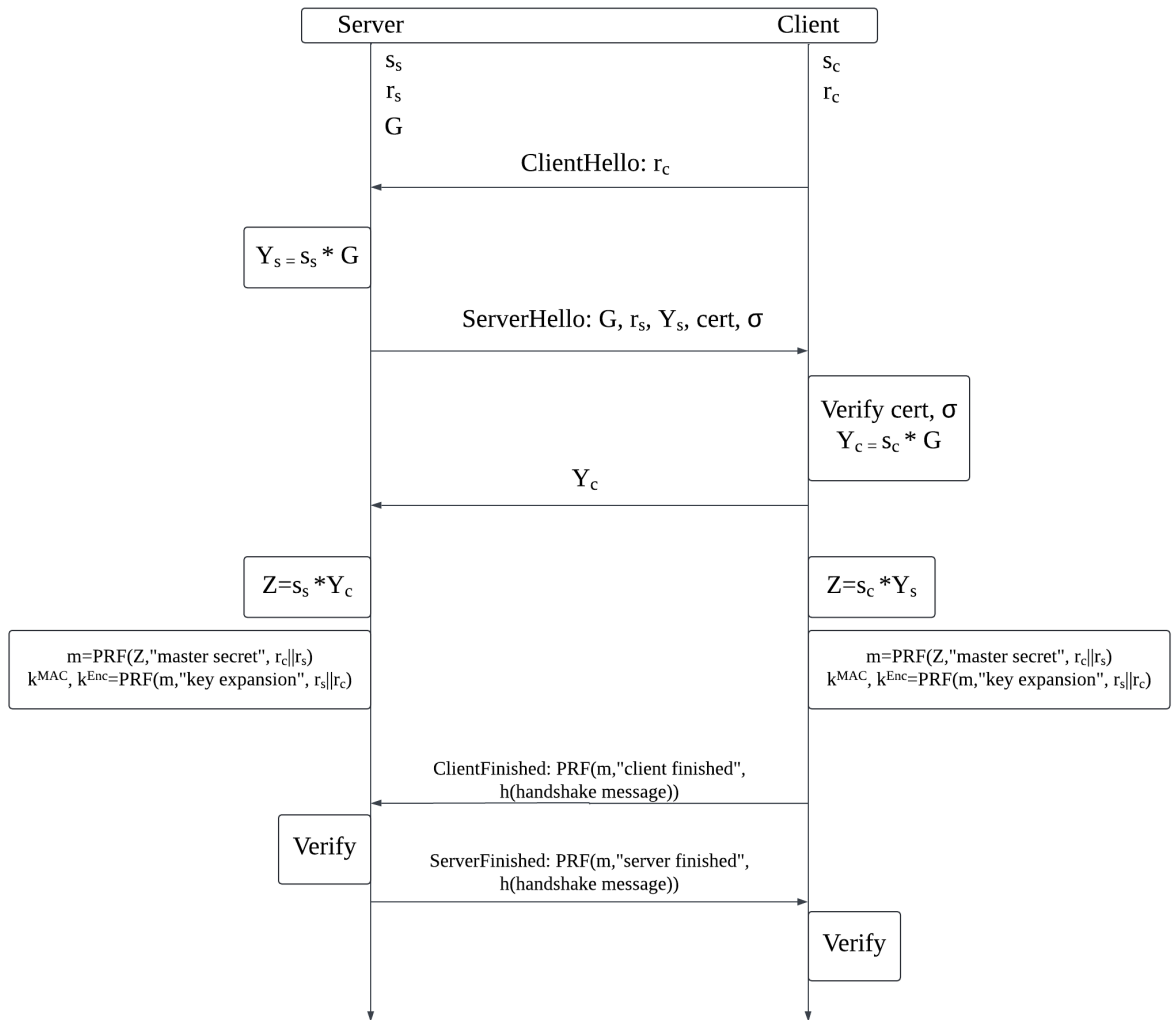


Figure A.1: TLS Under ECDHE

A.2 DECO Three-Party Handshake

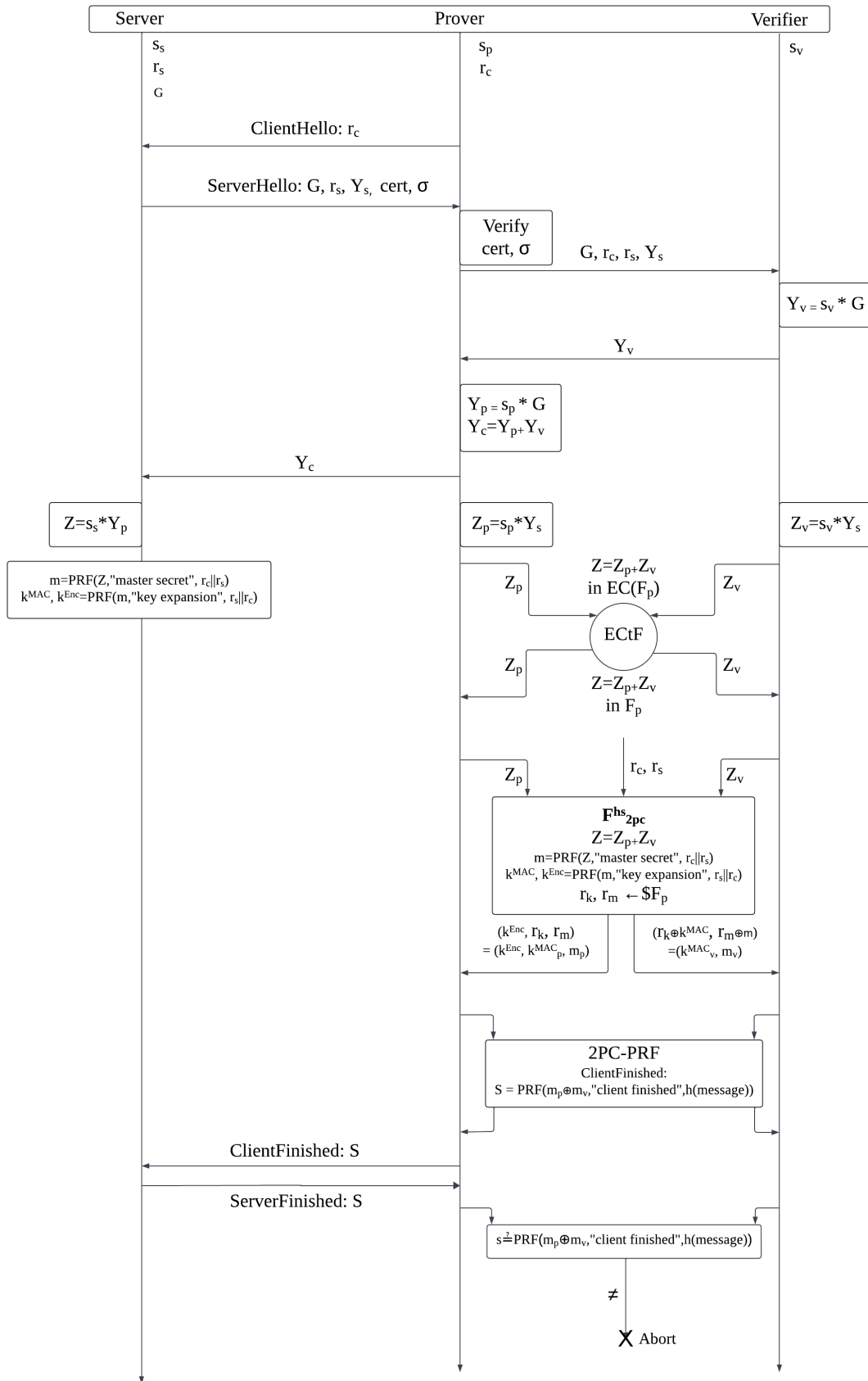


Figure A.2: Three-Party Handshake Under ECDHE

A.3 DECO Query Execution

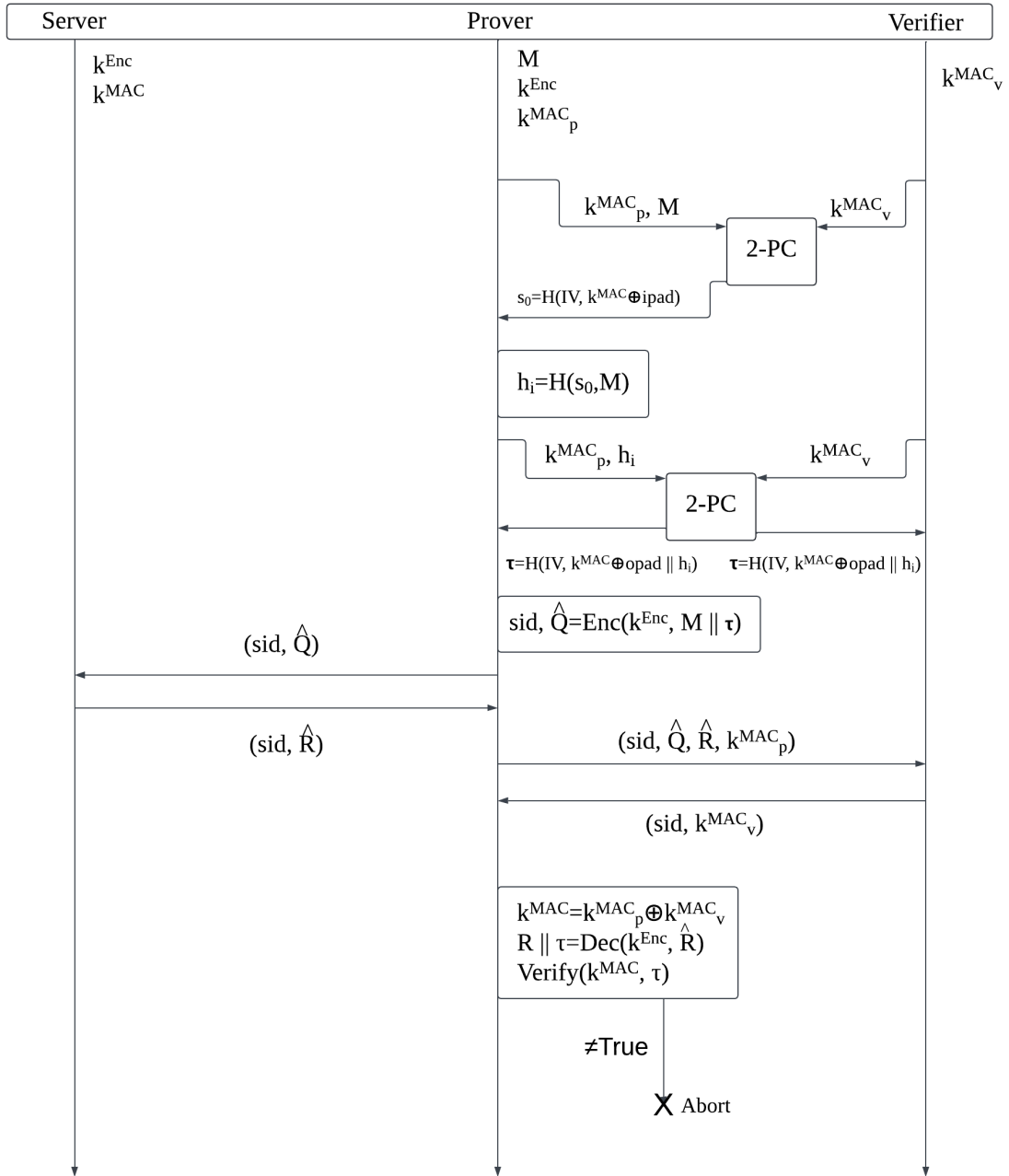


Figure A.3: Query Execution Under CBC-HMAC

A.4 Direct Approach

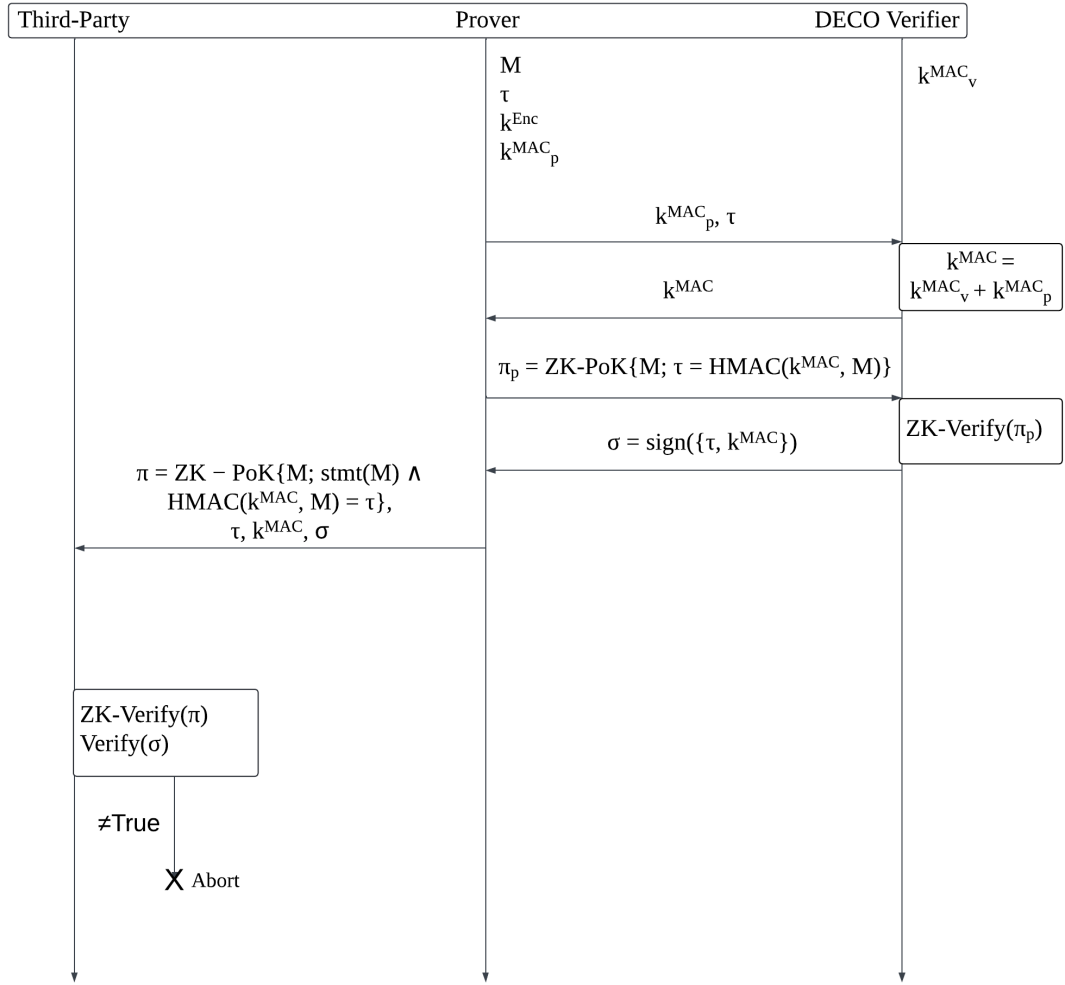


Figure A.4: Direct Proof on MAC Tag

A.5 Proof of Integrity on MAC Tag

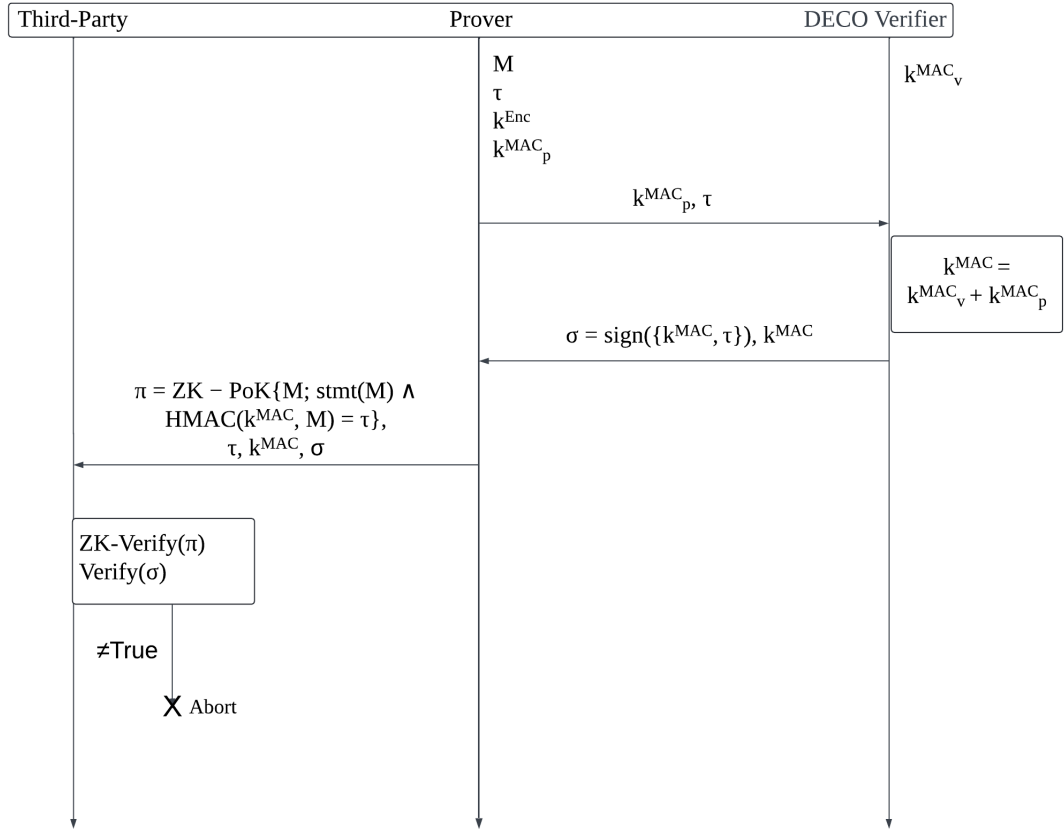


Figure A.5: Blind Proof on MAC Tag

A.6 Proof of Integrity on Merkle Trees

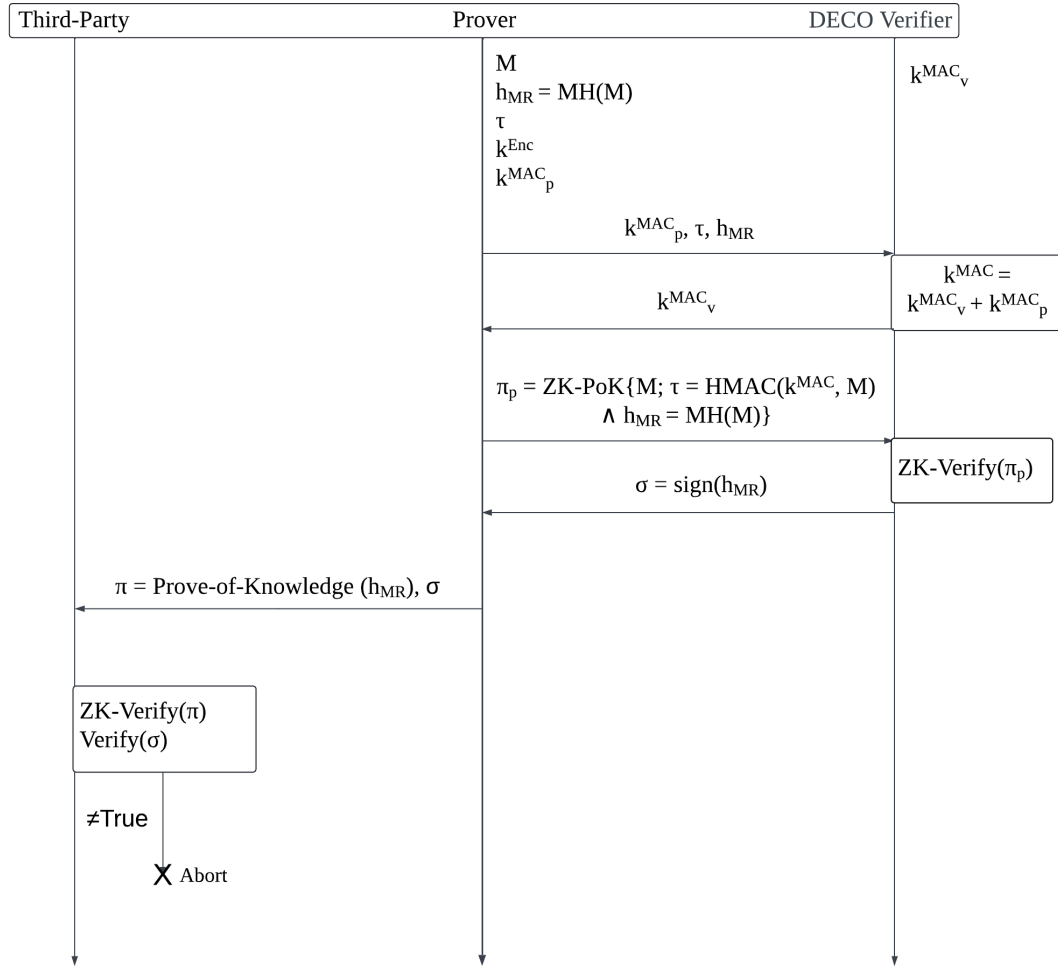


Figure A.6: Blind Proof of Integrity on Merkle Trees

A.7 Proof of Integrity on AES Encryption

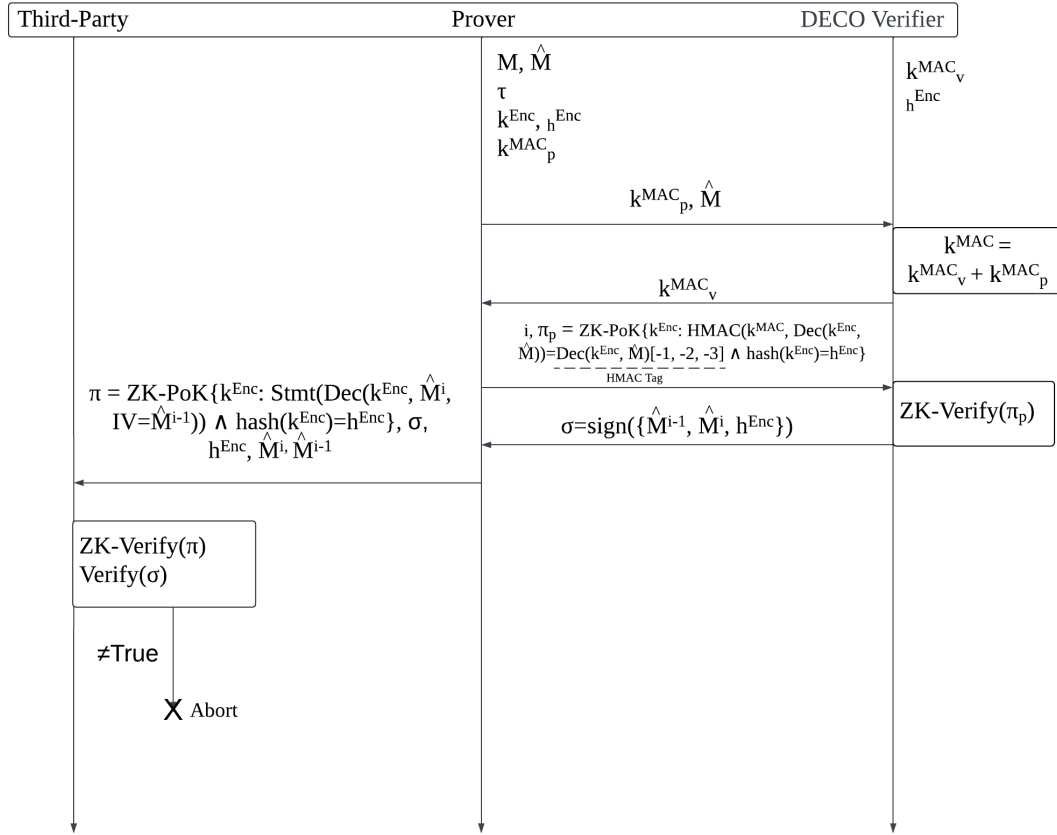


Figure A.7: Blind Proof of integrity on a Single AES Block

Appendix B

Blindness Security Proof

In this chapter, we will show that for any PPT adversary \mathcal{A} , we can construct a simulator Sim simulate the ideal world, and it is distinguishable with the real world for \mathcal{A} .

We only consider the case that \mathcal{A} corrupts the DECO Verifier or third party Verifier in case of the security of blindness. Therefore, we will construct ideal world simulator for the view of the DECO Verifier and third party Verifier for the 4 different approaches mentioned in chapter 4. Also, we will not look into the blindness property of DECO Three Party Handshake and query execution, therefore we will assume the ideal blindness security of Three Party Handshake and query execution, meaning except the output (share of MAC key, encryption key, MAC tag, etc) from the handshake protocol, each party learns nothing.

B.1 Blindness for the DECO Verifier

In this section, \mathcal{A} corrupts the DECO Verifier and wishes to extract more information through the communication. The idea of the proof is to construct a simulator for the DECO Verifier which simulate the ideal world. In ideal world, every message sent to the DECO Verifier does not contain any information related with the message M . If the ideal world simulated is indistinguishable with the real world for the DECO Verifier, the blindness property is guaranteed.

B.1.1 Direct Approach

At the end of three party handshake, the DECO Verifier hold his share of MAC key k_v^{MAC} . By assumption, Three Party Handshake and query execution are secure in terms of blindness, which means the DECO Verifier learns nothing else share of MAC key k_v^{MAC} . Also, we assume for this method, message M is high entropy. Given a real-world PPT adversary \mathcal{A} , simulator Sim does the following.

1. Sim runs \mathcal{A} , ideal zero knowledge proof functionality $ZK - PoK$ and ideal and DECO functionality (three party handshake and query execution) internally.
2. Sim calls the random oracle, generate a random bitstring r with the same length with M , and generate another random MAC key k^{MAC^0} with the same length with k^{MAC} . Sim compute the MAC key share of the Prover $k_p^{MAC^0} = k^{MAC^0} - k_v^{MAC}$.
3. Sim compute the MAC tag for the random message $\tau^0 = HMAC(k^{MAC^0}, r)$, and send τ^0 and $k_p^{MAC^0}$ to \mathcal{A} .
4. \mathcal{A} send back the complete MAC key $k^{MAC^0} = k_v^{MAC} + k_p^{MAC^0}$. Sim calls $ZK - PoK$ to generate the proof $\pi_p^0 = ZK - PoK\{r; HMAC(k^{MAC^0}, r) = \tau^0\}, r \leftarrow \{0, 1\}$ and send to \mathcal{A} . \mathcal{A} send back the signature and Sim output the signature.

To argue the ideal world execution is indistinguishable with the real world execution with \mathcal{A} , we will use several hybrid.

- **Hybrid H1** is the real world execution of Direct Approach method protocol mentioned in section 4.2.
- **Hybrid H2** is same as H1, except Sim simulate \mathcal{A} , ideal zero knowledge proof functionality $ZK - PoK$ and DECO functionality internally. Because of the property of a safe real world zero knowledge proof system (Completeness, Soundness, Zero-Knowledge) and safe assumption of DECO, also since the simulation of ideal functionality is perfect, H1 is indistinguishable with H2.
- **Hybrid H3** is same as H2, except Sim using a different random MAC key k^{MAC^0} and send $k_p^{MAC^0}$ instead of k_p^{MAC} . Since \mathcal{A} only holds half of the MAC key, and the key using for the MAC tag generation is $k^{MAC^0} = k_v^{MAC} + k_p^{MAC^0}$ which is the combination of \mathcal{A} 's share and $k_p^{MAC^0}$ send out, H2 indistinguishable with H3.

- **Hybrid H4** is same as H3, except *Sim* using a different random bitstring r with the same length with M . Since M is high entropy, by the property of HMAC function, $\tau^1 = \text{HMAC}(k^{\text{MAC}}, M)$ is indistinguishable with $\tau^0 = \text{HMAC}(k^{\text{MAC}^0}, r)$, therefore H3 is indistinguishable with H4.

In ideal world, the record \mathcal{A} saw is $R^0 = \{\pi^0, k^{\text{MAC}^0}, \tau^0\}$, which is exactly same as H4, therefore, H4 is indistinguishable with ideal world, which matches the equation in the definition mentioned in section 5.3.1. This proves that this method is safe in blindness.

B.1.2 MAC Tag Approach

In the last section, we have demonstrated the blindness for the DECO Verifier with records as $R^0 = \{\pi^0, k^{\text{MAC}^0}, \tau^0\}$. In the MAC tag method described in section 4.3, the record observed by the DECO Verifier is $R^0 = \{k^{\text{MAC}^0}, \tau^0\}$, which is identical to that in the direct approach, except it lacks the zero-knowledge proof. Consequently, it is trivial to conclude that in this method, the blindness of the DECO Verifier is maintained.

B.1.3 Merkle Tree Approach

Same as before in section B.1.1, assume M is high entropy, and assume the security in blindness of three party handshake and query execution, the DECO Verifier learns nothing but his share of MAC key k_v^{MAC} . Given a real-world PPT adversary \mathcal{A} , simulator *Sim* does the following.

1. *Sim* runs \mathcal{A} , ideal zero knowledge proof functionality $ZK - PoK$ and ideal and DECO functionality (three party handshake and query execution) internally.
2. *Sim* calls the random oracle, generate a random bitstring r with the same length with M , and generate another random MAC key k^{MAC^0} with the same length with k^{MAC} . *Sim* compute the MAC key share of the Prover $k_p^{\text{MAC}^0} = k^{\text{MAC}^0} - k_v^{\text{MAC}}$.
3. *Sim* compute the MAC tag for the random message $\tau^0 = \text{HMAC}(k^{\text{MAC}^0}, r)$, and compute the Merkle root of the random bitstring, which is $h_{MR}^0 = MH(r)$. Then send τ^0 , h_{MR}^0 and $k_p^{\text{MAC}^0}$ to \mathcal{A} .
4. \mathcal{A} send back the complete MAC key $k^{\text{MAC}^0} = k_v^{\text{MAC}} + k_p^{\text{MAC}^0}$. *Sim* calls $ZK - PoK$ to generate the proof $\pi_p^0 = ZK - PoK\{r : [\text{HMAC}(k^{\text{MAC}^0}, r) = \tau^0] \wedge [h_{MR}^0 = MH(r)]\}$, $r \leftarrow \{0, 1\}$ and send to \mathcal{A} . \mathcal{A} send back the signature and *Sim* output the signature.

To argue the ideal world execution is indistinguishable with the real world execution with \mathcal{A} , we will use several hybrid.

- **Hybrid H1** is the real world execution of Direct Approach method protocol mentioned in section 4.3.
- **Hybrid H2** is same as H1, except *Sim* simulate \mathcal{A} , ideal zero knowledge proof functionality $ZK - PoK$ and DECO functionality internally. Because of the property of a safe real world zero knowledge proof system (Completeness, Soundness, Zero-Knowledge) and safe assumption of DECO, also since the simulation of ideal functionality is perfect, H1 is indistinguishable with H2.
- **Hybrid H3** is same as H2, except *Sim* using a different random MAC key k^{MAC^0} and send $k_p^{MAC^0}$ instead of k_p^{MAC} . Since \mathcal{A} only holds half of the MAC key, and the key using for the MAC tag generation is $k^{MAC^0} = k_v^{MAC} + k_p^{MAC^0}$ which is the combination of \mathcal{A} 's share and $k_p^{MAC^0}$ send out, H2 indistinguishable with H3.
- **Hybrid H4** is same as H3, except *Sim* using a different random bitstring r with the same length with M . Since M is high entropy, by the property of HMAC function and Merkle hash function, $\tau^1 = HMAC(k^{MAC}, M)$ and $h_{MR}^1 = MH(M)$ is indistinguishable with $\tau^0 = HMAC(k^{MAC^0}, r)$ and $h_{MR}^0 = MH(r)$, therefore H3 is indistinguishable with H4.

In ideal world, the record \mathcal{A} saw is $R^0 = \{\pi^0, k^{MAC^0}, \tau^0, h_{MR}^0\}$, which is exactly same as H4, therefore, H4 is indistinguishable with ideal world, which matches the equation in the definition mentioned in section 5.3.1. This proves that this method is safe in blindness.

B.1.4 AES Approach

As mentioned in the first step in section 4.5, this method did some slight changes on three party handshake, by requiring the Prover to sent to the DECO Verifier on the hash of encryption key and initial vector. Therefore, different from the proof before, *Sim* will need to simulate this step in the ideal world. But we will still assume the blindness of DECO three party handshake and query execution. Thus, the message M in this method is not required to be high entropy, but the key is till need to be random. Also, the simulator will omit the initial vector and index that the DECO Verifier will signing on since it's trivial. Given a real-world PPT adversary \mathcal{A} , simulator *Sim* does the following.

1. *Sim* runs \mathcal{A} , ideal zero knowledge proof functionality $ZK - PoK$ and ideal and DECO functionality (three party handshake and query execution) internally.
2. During three party handshake, *Sim* generate the encryption key k^{Enc^0} from random with same length as k^{Enc} , *Sim* computing the hash $h^{Enc^0} = Hash(k^{Enc^0})$ then send to \mathcal{A} .
3. *Sim* runs the subsequent DECO functionality, till the end of query execution.
4. *Sim* calls the random oracle, generate a random bitstring r with the same length with M , and generate another random MAC key k^{MAC^0} with the same length with k^{MAC} . *Sim* compute the MAC key share of the Prover $k_p^{MAC^0} = k^{MAC^0} - k_v^{MAC}$.
5. *Sim* compute the MAC tag for the random message $\tau^0 = HMAC(k^{MAC^0}, r)$, and compute the ciphertext of the random bitstring, which is compute the ciphertext $\hat{M}^0 = Enc(k^{Enc^0}, r || \tau^0)$. Then send \hat{M}^0 and $k_p^{MAC^0}$ to \mathcal{A} .
6. \mathcal{A} send back his share of MAC key $k_v^{MAC^0}$ which $k_v^{MAC^0} = k^{MAC^0} - k_p^{MAC^0}$.
7. *Sim* generate the following proof and send to \mathcal{A} , \mathcal{A} send back the signature and *Sim* output the signature.:

$$\pi_p^0 = ZK - PoK\{k^{Enc^0}; [HMAC(k^{MAC^0}, Dec(k^{Enc^0}, \hat{M}^0)) = \underbrace{Dec(k^{Enc^0}, \hat{M}^0)[-1, -2, -3]}_{\text{HMAC tag } \tau^0}] \wedge [h^{Enc^0} = Hash(k^{Enc^0})]\}$$

To argue the ideal world execution is indistinguishable with the real world execution with \mathcal{A} , we will use several hybrid.

- **Hybrid H1** is the real world execution of Direct Approach method protocol mentioned in section 4.5.
- **Hybrid H2** is same as H1, except *Sim* simulate \mathcal{A} , ideal zero knowledge proof functionality $ZK - PoK$ and DECO functionality internally. Because of the property of a safe real world zero knowledge proof system (Completeness, Soundness, Zero-Knowledge) and safe assumption of DECO, also since the simulation of ideal functionality is perfect, H1 is indistinguishable with H2.
- **Hybrid H3** is same as H2, except *Sim* using a different random MAC key k^{MAC^0} and encryption key k^{Enc^0} , and send $k_p^{MAC^0}$, h^{Enc^0} instead of k_p^{MAC} , h^{Enc} . Since \mathcal{A} only holds half of the MAC key, and the key using for the MAC tag generation

is $k^{MAC^0} = k_v^{MAC} + k_p^{MAC^0}$ which is the combination of \mathcal{A} 's share and $k_p^{MAC^0}$ send out, also by the property of hash function on the random encryption key ($h^{Enc^0} = Hash(k^{Enc^0})$), H2 indistinguishable with H3.

- **Hybrid H4** is same as H3, except *Sim* using a different random bitstring r with the same length with M . Since \mathcal{A} cannot reverse the hash function and extract k^{Enc^0} from h^{Enc^0} , by the secrecy property of AES, \hat{M}^0 is indistinguishable with \hat{M}^1 , therefore H3 is indistinguishable with H4.

In ideal world, the record \mathcal{A} saw is $R^0 = \{\pi^0, k^{MAC^0}, h^{Enc^0}, \hat{M}^0\}$, which is exactly same as H4, therefore, H4 is indistinguishable with ideal world, which matches the equation in the definition mentioned in section 5.3.1. This proves that this method is safe in blindness.

B.2 Blindness for the third party Verifier

The proof of blindness for the third party Verifier is similar with the proof of the DECO Verifier. *Sim* simulating the ideal records of the Prover and DECO Verifier, generating a random message r except the part that containing the same statement and length as the real world (containing no information except the statement), and argue the record third party Verifier saw is indistinguishable with the real world record.

B.2.1 Direct Approach

Given a real-world PPT adversary \mathcal{A} , simulator *Sim* does the following.

1. *Sim* runs \mathcal{A} and ideal zero knowledge proof functionality $ZK - PoK$ internally.
2. *Sim* generate a random message r except the part that containing the same statement and length as the real world. *Sim* also generate a random MAC key k^{MAC^0} , compute the MAC tag $\tau^0 = HMAC(k^{MAC^0}, r)$ using the MAC key and random message.
3. *Sim* generate the zero knowledge proof $\pi^0 = ZK - PoK\{r; [stmt(r)] \wedge [HMAC(k^{MAC^0}, r) = \tau^0]\}$. *Sim* also compute a signature $\sigma^0 = sign(\tau^0)$ of MAC tag using a valid DECO Verifier secret key.
4. *Sim* send $\{\pi^0, \tau^0, k^{MAC^0}, \sigma^0\}$ to \mathcal{A} .

To argue the ideal world records is indistinguishable with the real world records with \mathcal{A} , we will use several hybrid.

- **Hybrid H1** is the real world records of Direct Approach method protocol mentioned in section 4.5.
- **Hybrid H2** is same as H1, except Sim simulate \mathcal{A} and ideal zero knowledge proof functionality $ZK - PoK$ internally. Because of the property of a safe real world zero knowledge proof system (Completeness, Soundness, Zero-Knowledge), also since the simulation of ideal functionality is perfect, H1 is indistinguishable with H2.
- **Hybrid H3** is same as H2, except Sim using a different random MAC key k^{MAC^0} , compute the corresponding MAC tag $\tau^0 = \text{HMAC}(k^{MAC^0}, M)$ and signature $\sigma^0 = \text{sign}(\tau^0)$, and send k^{MAC^0} , τ^0 , σ^0 instead of k^{MAC} , τ , σ . Since \mathcal{A} didn't know anything about the MAC key, and the key is signed by a valid DECO Verifier, H2 indistinguishable with H3.
- **Hybrid H4** is same as H3, except Sim using the random bitstring r with the same statement and length with M . Since by assumption message is high entropy, by the property of HMAC function, τ is indistinguishable with τ^0 , therefore H3 is indistinguishable with H4.

In ideal world, the record \mathcal{A} saw is $\{\pi^0, \tau^0, k^{MAC^0}, \sigma^0\}$, which is exactly same as H4, therefore, H4 is indistinguishable with ideal world.

B.2.2 MAC Tag Approach

The record in this method is same as direct approach in the previous section, therefore, the proof process is also the same as the previous section.

B.2.3 Merkle Tree Approach

Taking the method mentioned in section 4.4 6b as example. The record in this method is similar with the previous method. In this method, record of third party Verifier contain the Merkle root of the message, and the proof is about the Merkle root, and signature is on Merkle root. We know that the Merkle hash has the blindness property for high entropy input, therefore the Merkle root is indistinguishable with the ideal

world (with a random message r mentioned in previous section). Therefore, follow similar construction as above, it is trivial to conclude the blindness for third party Verifier.

B.2.4 AES Approach

In section B.1.4, we had prove the blindness for the Verifier with the record as $R^1 = \{\pi^1, k^{MAC^1}, h^{Enc^1}, \hat{M}^1\}$ is indistinguishable with $R^0 = \{\pi^0, k^{MAC^0}, h^{Enc^0}, \hat{M}^0\}$. In this method from the view of third party Verifier, the record is $R^1 = \{\pi^1, h^{Enc^1}, \hat{M}^1, \sigma^1 = \text{sign}(\hat{M}^1, h^{Enc^1})\}$. The private input for both zero knowledge proof is the encryption key k^{Enc} and signature will not leak any information except what it is signing on. Therefore, using the conclusion we got in section B.1.4, we can confidently conclude that the blindness for third party Verifier holds.