DECO: Liberating Web Data Using Decentralized Oracles for TLS - Summary Report

Bogdana Kolic
390123

Mina Petrovic
390255

Iman Attia
387653

ABSTRACT

Traditional Transport Layer Security (TLS) ensures data confidentiality and integrity but lacks a mechanism for proving the authenticity and provenance of accessed data to third parties. Without server-side modifications or trusted intermediaries this limitation restricts data portability and prevents secure data export. DECO (Decentralized Oracles for TLS)[3] addresses this issue by enabling users to prove the authenticity of TLS-protected data while preserving privacy through zero-knowledge proofs. Unlike existing solutions, DECO requires no modifications to web servers and supports standard TLS, making it a practical and secure approach for decentralized applications, smart contracts, and privacy-preserving verification systems.

1 INTRODUCTION

The Internet relies on Transport Layer Security (TLS) to provide secure communication, ensuring data confidentiality and integrity. However, TLS does not allow users to show to third parties that the data they access came from a legitimate source. This limitation restricts data portability, preventing its use in decentralized applications and trustless environments.

Existing solutions to this problem often require server-side modifications, rely on trusted hardware, or involve privacy risks. To address these challenges, DECO (Decentralized Oracle for TLS) offers a novel approach that enables users to securely export and verify TLS-protected data without requiring cooperation from web servers. Using cryptographic techniques such as zero-knowledge proofs, DECO ensures both authenticity and privacy, making it applicable to smart contracts, anonymous credential verification, and other privacy-preserving systems.

This report explores the design of DECO, its advantages over existing methods, and its potential impact on secure data portability, as well as applications implemented using DECO.

1.1 Why do we need DECO?

To understand the problem that DECO solves, consider a scenario involving Alice (the client and prover) and Charlie (the verifier). Charlie wants to sell his homework solutions but needs to ensure that the buyer has enough money before making the deal. Alice, who wants to purchase the solutions, logs into her bank account (Bob, the server) via a secure TLS connection. Bob, however, is unaware of Alice's intent and does not provide a way for her to prove her account balance to a third party.

Alice has a few options, but none are ideal. She could take a screenshot of her bank balance, but with the prevalence of photo-editing tools, Charlie has no way to verify its authenticity. She could share her bank credentials, but that would compromise her account security. Ultimately, there is no straightforward way for Alice to prove to Charlie that she has enough money without exposing sensitive information.

This is where DECO comes in. With DECO, Alice can cryptographically prove to Charlie that she has sufficient funds without revealing her exact balance or any other personal details. DECO ensures that the proof is verifiable and tamper-proof, solving the trust issue without requiring cooperation from the bank or compromising Alice's privacy.

[0]This report provides a summary on the ideas published by Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder and Ari Juels in [3] and presented at the *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*

2 BACKGROUND

2.1 Transport Layer Security (TLS)

The TLS protocol has two main components: *the handshake protocol* and *the record protocol*.

The handshake protocol is used to establish a connection between the client and the server: they decide which cryptographic algorithms (cipher suite) will be used, the server is authenticated, and a shared-secret is computed. DECO assumes the use of the elliptic curve Diffie-Hellman key exchange with ephemeral secrets **(ECDHE** [2]). This means that the client and the server use asymmetric cryptography to compute the shared secret, then derive the symmetric keys (one for encryption, one for the MAC tag) using a TLS-PRF key derivation function.

The record protocol is used for exchanging the application data. The plaintext data must first be split into fixed-sized fragments called ***records***. Then, the MAC tag of each record is computed, and encrypted together with the data before transmission. DECO supports the **AES**cipher with either the **CBC-HMAC** mode or **GCM**.

2.2 Multi-party computation

Multi-party computation protocols allow n parties $P_1, ...P_n$, each holding a secret s_i to jointly compute some $f(s_1, ...s_n)$, without revealing any of the secrets s_i to $P_{j\neq i}$. DECO uses the special-case two-party computation protocols **(2PC)**. In this scenario, $n = 2$, the protocols are secure if they do not leak any secrets even if an adversary corrupts one of the two parties.

3 THE *DECO* PROTOCOL

3.1 Problem statement

The goal of the paper is to construct "oracles" - entities that can prove provenance and properties of online data [3]. We consider a prover P, a verifier V and a server S. The prover P can obtain some private data from the server S, potentially by providing a secret input θ_S. P wishes to prove to V that the data came from S. In addition, it might also want to prove some statements about data.

The goal is that these proofs happen in zero-knowledge, keeping both the data and the optional secret input hidden from the verifier.

One solution would be to let the server act as an oracle - thus being a central point of control. DECO, in contrast, implements *decentralized oracles*: anyone can prove the provenance of the data she can access, without requiring neither trusted hardware nor any server-side modifications. The ideas are presented on the example of TLS 1.2 using CBC-HMAC, but can be modified to support TLS version 1.3 and AES-GCM cipher suite. For more details, we refer the reader to [3].

Threat model. DECO assumes the presence of a static, malicious network adversary. The server S is assumed to be honest, and all of its responses are considered to be the ground truth. DECO aims to provide the following security guarantees (as defined in [3]):

- *Prover-integrity*: A malicious P cannot forge content provenance, nor can she cause S to accept invalid queries or respond incorrectly to valid ones.
- *Verifier-integrity* A malicious V cannot cause P to receive incorrect responses
- *Privacy* A malicious V learns only public information (Query,S) and the evaluation of Statement(R), where Query is the query template and R is the server's response.

3.2 Overview of *DECO*

A strawman protocol. A naive implementation that seemingly achieves the goal would be the following strawman protocol: P establishes the connection to S, sends the queries and receives the responses from S, and keeps a record of all exchanged encrypted messages $\hat{M} = (\hat{Q}, \hat{R})$, where $\hat{Q} = (\hat{Q_1}, ...\hat{Q_n})$ are the queries and $\hat{R} = (\hat{R_1}, ...\hat{R_n})$ are server's responses. Then, P computes in zero-knowledge the proof p_r that each received response decrypts to a plaintext record and its correct MAC tag with respect to the key k^{MAC}, as well as that the evaluation of the desired statement on the response is $b = Statement(R)$. P also computes p_q, the proof that the query Q was generated by using the correct template on its private input θ_s, that is, $Q = Query(\theta_s)$. Finally, P sends $(p_q, p_r, k^{MAC}, \hat{M}, b)$ to V.
Although $\hat{M}$ seems to be a secure commitment to the session data (the CBC-HMAC binds to the underlying plaintext data), the prover integrity does not hold - the authenticity is not ensured because in TLS the server and the client both have the same key, hence P can use that key to to insert arbitrary encrypted data in $\hat{M}$. Moreover, the computational cost of generating p_r and p_q is high, and should be reduced to allow the protocol to be used in practice.

DECO. DECO solves the binding problem by introducing the ***three-party handshake***: during the TLS handshake, P and V each receive their share of the session key k^{MAC}, and V reveals its share only after P has committed to the data. The protocol is implemented only on the prover and receiver side, from the server's perspective, it performs a regular TLS handshake with P.
Since the query needs to be sent to S before V reveals its share of k^{MAC}, DECO relies on 2PC to allow P and V to jointly compute the correct tag of the query, but introduces custom optimizations to reduce the computational cost of ***query execution***.

Proof generation. Once the prover commits to a ciphertext, its goal is to prove statements about it to V. The simplest way to do so is to share the encryption key with V. However, this would not preserve privacy. Another approach P could take is using generic zero-knowledge proofs - but is expensive.
DECO introduces ***selective opening*** and ***two-stage parsing*** to counter this. With selective opening, P can either reveal only a chunk of the plaintext message to V - ***Reveal mode***, or reveal everything but some chunks of the plaintext message - ***Redact mode***. Still, this is not enough. P needs to prove that the revealed string appears in the right context in R - ***context integrity***. Consider the following example: Alice wants to prove to Charlie that she has over $5000 in her account, but her bank statement contains a substring "balance": $1000. Without context checking, Alice could send a message containing a substring "balance": $6000 to the server in the same TLS session and wait for the response. Then, she would use the same substring in the response to falsely prove a to Charlie that she has enough money on her account. To optimize the cost of zero-knowledge proofs of ontext integrity, DECO first pre-processes the response and then proves in zero-knowledge that the revealed string appears in the right context - this is the two-stage parsing.

3.3 Three-party handshake

DECO's three-party handshake (3P-HS) allows P and V to act as one client to the server S, providing a way for P to commit to the session data without requiring any changes to the server's code. At the end of the protocol, P and S hold the same encryption key k^{ENC}, while P and V receive secret shares of the MAC key, k_P^{MAC} and k_V^{MAC}, so that the server's key k^{MAC} is equal to $k_P^{MAC} + k_V^{MAC}$.
In ECDHE, we are working with points on the elliptic curve $EC(\mathbb{F}_p)$, generated by a point G. The handshake begins when P sends a standard TLS **ClientHello** message to S. S responds with a **ServerHello** message and a signed ephemeral DH public key $Y_S = s_S \cdot G$. P verifies that the certificate from the ServerHello message is valid and checks the signature, then it forwards the message to V to perform the same checks. If everything is correct, V computes $Y_V = s_V \cdot G$ from its sampled secret s_V and sends it to P. P then uses its secret s_P to compute $Y_P = s_P \cdot G + Y_V$ and sends it to S, which uses it to compute the shared secret Z as $s_S \cdot Y_P$. As we want $Z = Z_P + Z_V$, the prover and the verifier compute their shares as $Z_P = s_P \cdot Y_S$ and $Z_V = s_V \cdot Y_S$, respectively. Notice that:

(1) This protocol is secure as long as the discrete logarithm is hard in $EC(\mathbb{F}_p)$, and
(2) Z stays hidden from both P and V.

The second step of the 3P-HS is the key derivation - P and S use a 2PC protocol to compute their additive shares of the key from their secrets Z_P and Z_V. This entails performing both arithmetic operations (to add points in $EC(\mathbb{F}_p)$), and bitwise operations (to evaluate the TLS-PRF) function, which incurs a high cost. Since the TLS-PRF function is evaluated on the x-coordinate of $Z \in \mathbb{F}_p$, and addition of points in $\mathbb{F}_p$ costs a lot less than addition of points in $EC(\mathbb{F}_p)$, DECO proposes an intermediary step in the key derivation process: P and V first use an $ECtF$ function [3] to convert their

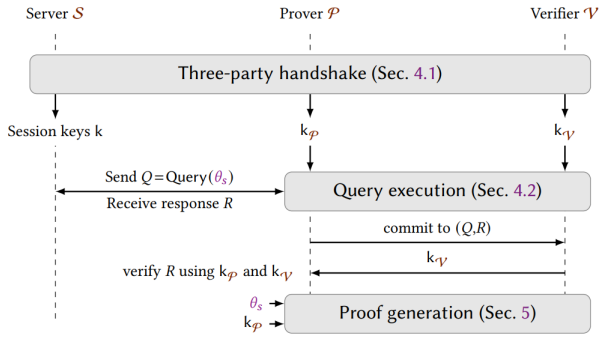

Figure 1: The three phases of DECO [3]

shares of the secret Z in $EC(\mathbb{F}_p)$ into shares of Z's x-coordinate in $\mathbb{F}_p$. Once the $ECtF$ function is executed, P and V evaluate the TLS-PRF in 2PC.

3.4 Query execution

After the three-party handshake, P and V have their secret shares k_P^{MAC} and k_V^{MAC} of the key k^{MAC}, which they use to jointly compute the MAC tag of the query in 2PC, before P can send it to S. However, computing the tag of a message m using the key k in 2PC according to the formula $HMAC_H(k,m) = H((k \oplus opad)||H((k \oplus ipad)||m))$, where H represents SHA-256, can be expensive for large messages. In case of a large query, most of the complexity lies in computing the inner hash $H((k \oplus ipad)||m)$, as the outer hash is computing the hash of the inner hash - a substantially shorter message. The idea for the optimization is to leverage the Merkle–Damgård scheme, and primarily perform the computations locally at P, without using the 2PC. If we denote f_H the one-way compression function of H, the computation of the inner hash becomes $H((k \oplus ipad)||m) = f_H(f_H(IV, k^{MAC} \oplus ipad), Q)$, with IV the initialization vector. Now, as f_H is one-way, the key k^{MAC} cannot be recovered from $f_H(IV, k^{MAC} \oplus ipad)$, and there is no reason to keep using the 2PC to compute the inner hash. We just need to provide $f_H(IV, k^{MAC} \oplus ipad)$ to P and it can finish the computation. Then, we use 2PC to compute the outer hash on a smaller input.

3.5 Full protocol

When P receives all the responses from the server, it can send $\hat{M}$ to V as a commitment, and V will respond by revealing its share of the MAC key. With the full key k^{MAC}, P can then verify the integrity of S's responses and start proving statements about them. Figure 1 shows all three phases of DECO.

3.6 Selective opening

The efficiency of selective opening depends on the cipher suite used in the TLS session: DECO can reveal chunks of a message at a TLS record-level in CBC-HMAC, and redact them at an AES block-level. With GCM, both revealing and redacting is efficient at a block level. We show how DECO implements the *Reveal mode* with CBC-HMAC, the other examples can be found in [3].

Revealing a TLS record. Proving the correct encryption of a message M into $\hat{M}$ without sharing the encryption key k^{ENC} can be done by proving the correct encryption of each AES block (up to 1027 blocks) in zero knowledge. To optimize this in CBC-HMAC mode, DECO lets P prove in zero knowledge the encryption of the last 3 blocks corresponding to the MAC tag, then reveal the entire plaintext record to V. This scheme is secure if the hash-function in HMAC is collision-resistant.

4 APPLICATIONS

DECO can be used in any oracle-based application. To demonstrate its versatility, the authors implemented and evaluated three use cases:

(1) A confidential financial instrument using smart contracts.
(2) Anonymous credential conversion from legacy credentials.
(3) Privacy-preserving price discrimination reporting.

4.1 Confidential financial instruments

Financial derivatives, such as binary options, are commonly implemented in smart contracts and rely on authenticated data feeds like stock prices. A binary option is a contract between two parties wagering on whether the price P^* of an asset N will meet or exceed a target price P at a future time (e.g., the close of day D). A smart contract can call an oracle O to determine the outcome.

Traditionally, the oracle learns the details of the financial instrument, including N and P. DECO enables secure execution of binary options without revealing these details to the oracle. Unlike previous methods that required Trusted Execution Environments (TEEs), DECO preserves privacy by processing option data off-chain and reporting only the outcome.

The concept is that the winner of the binary option acts as $\mathcal{P}$ and receives a signed statement, *Stmt*, from $\mathcal{O}$, who serves as the verifier $\mathcal{V}$. We now outline the protocol and how it functions.

Protocol. Let $\{sk_{\mathcal{O}}, pk_{\mathcal{O}}\}$ represent the oracle's key pair. In this protocol, a binary option is defined by three parameters: the asset name N, the threshold price P, and the settlement date D. A message M is committed as $C_M = \text{com}(M, r_M)$ using a random witness r_M.

(1) **Setup:** Alice and Bob mutually agree on the binary option parameters $\{N, P, D\}$ and deploy a smart contract SC identified by ID_{SC}. This contract includes $pk_{\mathcal{O}}$, the parties' addresses, and commitments to each parameter $\{C_N, C_P, C_D\}$, along with witnesses known to both participants. They also establish the public parameters θ_p (such as the URL to fetch asset prices).
(2) **Settlement:** If Alice wins the option, she must prove her claim. To do so, she utilizes DECO to construct a zero-knowledge proof (ZK proof) demonstrating that the asset price retrieved supports her position. She and $\mathcal{O}$ (the verifier) execute the DECO protocol to obtain the asset price from θ_p. It is assumed that the response includes (N^*, P^*, D^*). Along with

the DECO ZK proof that confirms the source θ_p. Upon successful proof verification, the oracle returns a signed copy. If the proof verification succeeds, the oracle generates a digitally signed message that includes the contract ID, denoted $S = \text{Sig}(\text{sk}_{\mathcal{O}}, \text{ID}_{SC})$.

(3) **Payout:** Alice sends the signed message S to the smart contract. Upon validating the signature, the contract releases the payment to the designated winner.

4.2 Legacy credentials to anonymous credentials: Age proof

User credentials are typically confined within the service provider's environment. While some providers offer third-party access through OAuth tokens, these tokens often expose user identities. DECO enables users with existing credentials—referred to as legacy credentials—to prove specific statements about their credentials to external verifiers, all while preserving anonymity. DECO is the first system to transform any web-based legacy credential into an anonymous one, without needing backend server modifications or trusted hardware.

As a use case, consider a student wishing to prove they are over 18 using demographic data stored on their university's website. This age verification can then be shared with third parties such as a government agency for issuing a driver's license or a healthcare provider for medical consent. This scenario is implemented using the AES-GCM cipher suite and a two-stage parsing approach, optimized using unique key identifiers.

4.3 Price discrimination

Price discrimination refers to the practice of offering the same product or service at varying prices to different customers. With widespread consumer tracking, many e-commerce and booking platforms employ advanced forms of price discrimination—for example, tailoring prices based on a buyer's zip code. Although this strategy can enhance economic efficiency and is generally lawful, it faces regulatory scrutiny in certain regions. In the United States, the FTC prohibits price discrimination when it causes competitive harm, and new privacy-focused regulations in Europe, such as the GDPR, are also questioning its legality. Regardless of legality, most consumers are uncomfortable with being targeted by price discrimination.

At present, there is no reliable method for users to report incidents of online price discrimination. DECO addresses this by enabling users to prove that the price they were shown for a product exceeds a certain threshold—without revealing private details like their name or address. This claim is cryptographically verifiable. We implement this scenario using the AES-GCM cipher suite for the TLS session and selectively disclose 24 AES blocks that include the relevant order information and the request URL.

5 IMPLEMENTATION

The implementation of DECO involved developing the three-party handshake protocol (3P-HS) for TLS 1.2 and query execution protocols (2PC-HMAC and 2PC-GCM) in approximately 4,700 lines of C++ code. The system used a hand-optimized TLS-PRF circuit with an AND complexity of 779,213, along with AES circuits from previous work. Cryptographic operations were supported by Relic for the Paillier cryptosystem and the EMP toolkit for maliciously secure two-party computation (2PC) protocols. Integration with mbedTLS, a widely used TLS implementation, was achieved for 3P-HS and 2PC-HMAC, while 2PC-GCM required additional engineering effort.

For proof generation, DECO employed zero-knowledge proofs (ZKPs) using libsnark [1], chosen for its mature tooling and efficient verification. Developers could adapt statement templates for specific applications using SNARK compilers such as xjsnark, which abstracted low-level circuit details.

6 EVALUATION

Performance evaluations were conducted in both LAN and WAN settings. In the LAN environment, DECO demonstrated high efficiency, with the three-party handshake completing in 0.37 seconds and 2PC-HMAC processing records in 0.13 seconds each. The cost of 2PC-GCM varied with query size, ranging from 36.65 ms for 256B requests to 204.7 ms for 2KB requests. In the WAN setting, network latency became the dominant factor, increasing the online phase of 3P-HS to 2850 ms. Despite this, the performance remained acceptable for applications where DECO would be used periodically rather than in real-time.

Proof generation metrics were measured across three applications: Binary Option, Age Proof, and Price Discrimination. The Binary Option application, the most complex, required around 12.97 seconds for proof generation, 617000 arithmetic constraints, and 1.78 GB of memory, with proofs sized at 861 B. Verification times were consistently fast, taking no more than 0.05 seconds. While DECO is slower than trusted hardware solutions like Town Crier (which completes similar tasks in 0.6 seconds), it provides stronger cryptographic guarantees without relying on trusted execution environments.

7 CONCLUSION

By eliminating the need for server-side cooperation and enabling secure data export, DECO provides a significant improvement over existing oracle-based approaches. Its compatibility with standard TLS and its ability to facilitate trustless verification make it a powerful tool for decentralized and federated systems, where data authenticity and integrity are crucial.

REFERENCES

[1] 2025. libsnark: A C++ library for zk-SNARKs. (2025). https://github.com/scipr-lab/libsnark Accessed: 2025-03-28.

[2] Bodo Moeller, Nelson Bolyard, Vipul Gupta, Simon Blake-Wilson, and Chris Hawk. 2006. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492. (May 2006). https://doi.org/10.17487/RFC4492

[3] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. 2020. Deco: Liberating web data using decentralized oracles for tls. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1919–1938.