DIDO: Data Provenance from Restricted TLS 1.3 Websites

The extended version

Kwan Yin Chan, Handong Cui, Tsz Hon Yuen

The University of Hong Kong
{kychan, hdcui, thyuen}@cs.hku.hk

Abstract. Public data can be authenticated by obtaining from a trustworthy website with TLS. Private data, such as user profile, are usually restricted from public access. If a user wants to authenticate his private data (e.g., address) provided by a restricted website (e.g., user profile page of a utility company website) to a verifier, he cannot simply give his username and password to the verifier. DECO (CCS 2020) provides a solution for liberating these data without introducing undesirable trust assumption, nor requiring server-side modification. Their implementation is mainly based on TLS 1.2.
In this paper, we propose an optimized solution for TLS 1.3 websites. We tackle a number of open problems, including the support of X25519 key exchange in TLS 1.3, the design of round-optimal three-party key exchange, the architecture of two-party computation of TLS 1.3 key scheduling, and circuit design optimized for two-party computation. We test our implementation with real world website and show that our optimization is necessary to avoid timeout in TLS handshake.

Keywords: TLS1.3, two-party computation, decentralized identification oracle

1 Introduction

Fact-checking over public information can be done by verifying the data from a trustworthy website. By retrieving a news article from a trusted news website with TLS, one can ensure that the information comes from a legitimate source and it is not altered. However, it is not easy to obtain the same security guarantee for data with restricted access. Suppose Alice wants to apply for a deposit account in an online bank (which does not have any physical branch) and she needs to provide an address proof. The photo of her letter or the PDF of her utility bill may be digitally edited. It is also not feasible to ask Alice to provide her username and password of her online utility account for validating her address. If Alice logins to her online utility account and then forwards the encrypted HTML page returned by the server to the bank, the bank will not accept this proof. It is because TLS only provides authenticity and data integrity to the client only, but not towards any third party. The session key (derived from TLS handshake) used to authenticate the HTML page is known to Alice and hence she is able to modify it.

In general, user's private data is often locked up by data owner. There is a strong demand for providing data provenance over such restricted information. There are a number of existing solutions with different limitations. Some of them rely on trusted hardware [26], but there are various attacks on these hardware [15]. Some solutions require the change of the server setup, such as requiring the server to install TLS extension [18], or changing the application-layer logic [6, 25]. These solutions are not compatible with existing TLS websites. In order to provide a generic solution with backward compatibility, the ideal case is to propose a method that does not require any modification from the server side nor any hardware requirement.

1.1 Decentralized Oracles for TLS

TLSNotary [1] proposed an architecture that allows a prover to provide irrefutable evidence to a third party (the verifier) that certain web traffic occurred between himself and a server. As shown in Fig. 1, the prover and the verifier jointly acts as a client and the process is transparent to the server. TLSNotary works with the depreciated TLS 1.0 and 1.1. In the key exchange phase, the prover and the verifier jointly run the RSA key exchange algorithm and obtain a share of key k_P and k_V respectively. By using two-party computation (2PC), they can be used to derive the

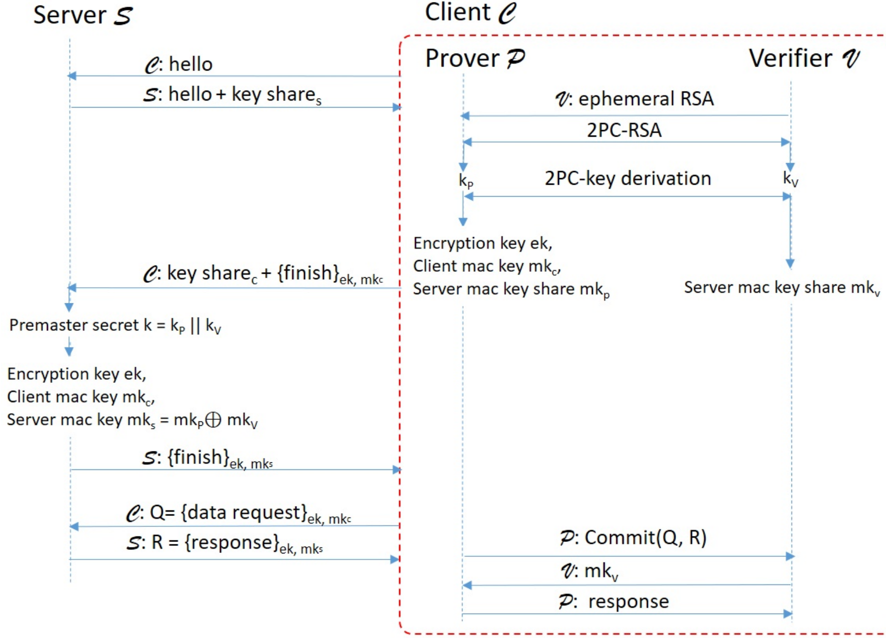

Fig. 1: Overview of the system architecture in TLSNotary. || stands for bit-wise concatenation, $\oplus$ stands for XOR, and $\{m\}_{\mathsf{ek},\mathsf{mk}}$ stands for the encryption of m using the symmetric key ek and mac key mk.

encryption key ek and the client MAC key mk_C for the prover. The prover and the verifier also get their own shares of the server MAC key mk_P and mk_V, such that the final server MAC key $\mathsf{mk}_S = \mathsf{mk}_P \oplus \mathsf{mk}_V$. The prover sends his encrypted data request by using ek and mk_C. When the server returns an encrypted response R, the prover commits it to the verifier, and the verifier returns mk_V to the prover. The prover reconstructs the final server MAC key $\mathsf{mk}_S = \mathsf{mk}_P \oplus \mathsf{mk}_V$ and decrypts the traffic. The prover passes the decrypted traffic to the verifier for validation. Since the prover does not process the entire mk_S before commitment, the verifier can be sure that the traffic data is authentic.

Zhang *et al.* [27] formalized the notion of *decentralized oracle*, which provide *authenticity* and *privacy assurances* to Internet data from any website running standard TLS. They proposed a decentralized oracle protocol DECO, which used TLSNotary's high-level architecture with the adoption of TLS 1.2 for data authenticity, and also provided privacy protection for the decrypted traffic with the use of zero-knowledge proof, as shown in Fig. 2. Firstly, DECO used a three-party handshake for Elliptic-Curve Diffie-Hellman Ephemeral (ECDHE) using the curve secp256r1. The prover and the verifier compute additive shares of the ECDHE session key k (by a protocol called ECtF). Then, they derive secret-shared session keys (master secret, encryption keys, mac keys) by securely evaluating the HMAC-SHA256 function by using 2PC-SHA256. In particular, the master secret and the mac keys are shared while the prover holds the encryption keys. Afterwards, the prover prepares an encrypted request Q', runs a 2PC-HMAC with the verifier to compute an HMAC tag τ with the client mac key, and sends the request $Q = Q'||\tau$ to the server. Finally, the prover receives an encrypted response from the server and commits it to the verifier. The verifier returns his share of the server mac key. The prover decrypts the response to obtain R and a tag τ'. The tag τ' is verified with the reconstructed server mac key mk_S. Finally, the prover uses a zero-knowledge proof SNARK [3] to show that the response R is correctly decrypted and verified by τ', and it satisfies some relation. The prover can also selectively open partial information of R.

1.2 Motivation: Compatibility with TLS 1.3

TLS 1.2 was standardized in 2008. Many of the major vulnerabilities in TLS 1.2 is caused by the use of older cryptographic algorithms that were still supported. For example, the CBC mode ciphers suffer from the BEAST [9] and Lucky13 [2] attacks. TLS 1.3 was published in 2018. It drops

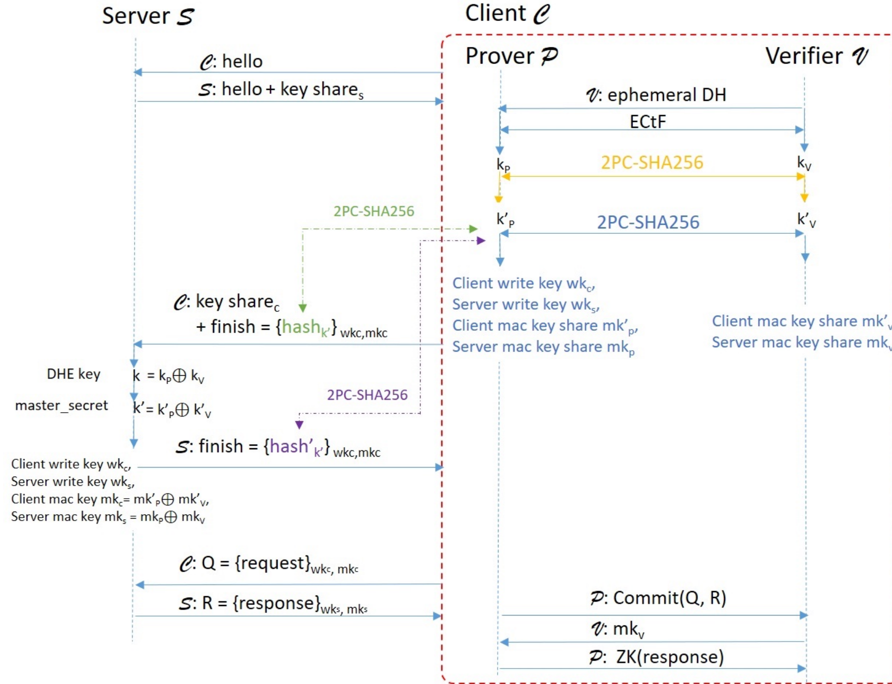

Fig. 2: Overview of the system architecture in DECO. $\oplus$ stands for XOR, and $\{m\}_{\mathsf{wk},\mathsf{mk}}$ stands for the encryption of m using the symmetric key wk and mac key mk.

support for these vulnerable cryptographic algorithms, simplifies the selection of cipher suites, and is faster than TLS 1.2. According to a recent survey[1], TLS 1.3 becomes the preferred TLS protocol for 63% of the top one million web servers on the Internet in 2021.

During the TLS handshake stage, the hello messages and the key shares are sent in plaintext, while the finish messages (including the HMAC value) are encrypted by the derived keys (as shown in Fig. 1, 2 and 3). Most servers have a specific timeout value on the *TLS handshake timeout*. The default value is usually around 10-15 seconds[2]. It is challenging to finish the 2PC within the time limit, due to the complexity of the key derivation.

There are some changes in TLS 1.3 that makes it difficult to build a decentralized oracle for TLS 1.3, namely the changes in TLS 1.3 key scheduling and the X25519 key exchange protocols.

TLS 1.3 Key Scheduling. In TLS 1.2, the DHE key is taken as the input key of the HMAC-SHA256 function to derive a master secret. The master secret is then used to (1) compute the HMAC for the client and server finish messages, and (2) derive all encryption keys (called *write keys* in the RFC 5246 [7]) and mac keys using one HMAC. Hence, a total of four 2PC-HMAC is needed. In DECO, a 2PC-HMAC is computed by one invocation of 2PC-SHA256 and it takes around 2.5 seconds for the WAN setting [27]. Hence, the 2PC-HMAC part already uses 10 seconds in the TLS 1.2 handshake.

In TLS 1.3, the key scheduling is much more complicated (as shown in Fig. 5) than the TLS 1.2 version (Fig. 6). Starting from the DHE key, it requires five HMAC-SHA256 (used in $\mathsf{HKDF.Extract}$ and $\mathsf{HKDF.Expand}$) to generate the handshake keys $\mathsf{tk}_{\mathsf{chs}}$ and $\mathsf{tk}_{\mathsf{shs}}$, and an extra six HMAC-SHA256 to generate the application encryption keys $\mathsf{tk}_{\mathsf{capp}}$ and $\mathsf{tk}_{\mathsf{sapp}}$. Hence, the 2PC of the TLS 1.3 key scheduling is much more complicated and time consuming.

X25519 Key Exchange. TLS 1.2 supports a number of different elliptic curves for ECDHE. TLS 1.3 simplified that to five curves, in which X25519 and X448 (ECDHE over Curve25519 and Curve448) are newly added to the standard. They provide better protection against side-channel attacks, and have clear explanation on the choice of their parameters. It is not easy to integrate DECO with these new curves for two reasons. Firstly, DECO [27] was designed to support

[1] https://www.f5.com/labs/articles/threat-intelligence/the-2021-tls-telemetry-report
[2] E.g., https://www.ibm.com/docs/en/zos/2.5.0?topic=considerations-handshake-timer

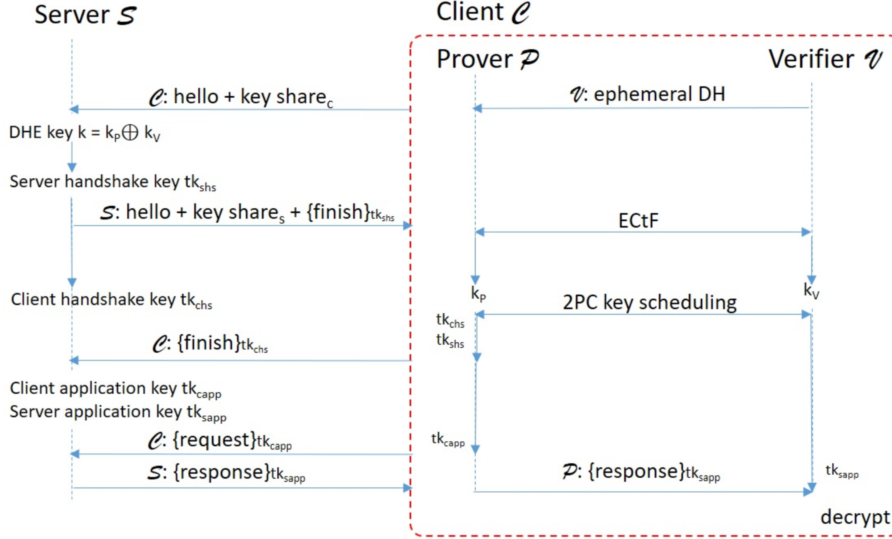

Fig. 3: Modified system architecture for TLS 1.3 (without selective opening). The symbol $\{m\}_k$ stands for the symmetric key authenticated encryption of m using the key k.

ECDHE over the elliptic curve of the form $Y^2 = X^3 + a_1X + a_0 \bmod p$ (e.g., secp256r1, secp384r1 and secp521r1 in TLS 1.3). Their ECtF does not support the Montgomery curve of the form $Y^2 = X^3 + a_2X^2 + a_1X \bmod p$ (e.g., Curve25519 and Curve448 in TLS 1.3), which is also widely used in practice. Secondly, the API for X25519 and X448 defined in RFC 7748 only performs a masked scalar multiplication and returns the x-coordinate. Only the x-coordinate is sent during the TLS 1.3 handshake for X25519 and X448. Hence, existing TLS libraries may not have the API for accessing the y-coordinate, computing point addition and scalar multiplication, which are required for running the three-party handshake protocol in DECO. Rewriting the underlying elliptic curve library can be error-prone.

1.3 Our Contributions

In this work, we construct a practical decentralized oracle for TLS 1.3. In particular, we work on improving the three-party handshake protocol, and design our own 2PC key scheduling for TLS 1.3.

For three-party handshake using ECDHE, we first modify the ECtF protocol to support the Curve25519 and Curve448. The original ECtF protocol in DECO has 8 rounds of communication between the prover and the verifier. The total round-trip time of ECtF (4×67ms $= 268$ms in [27]) already contributes to 9.4% of the online running time of the handshake in DECO. We propose a round-optimal ECtF protocol to reduce the total round complexity from 8 to 3. We achieve this by designing our dedicated multiplicative-to-additive (MtA) protocol, instead of using the existing MtA protocol as in DECO [27]. Details can be found in section 4.1.

During our implementation, we solved the API problem for running the ECtF protocol with X25519. In particular, we make use of the API for EdDSA in TLS 1.3. We observe that the twisted Edward curve for Ed25519 is equivalent to the Curve25519. Hence, we need to do a coordinate conversion and use the elliptic curve arithmetic API provided in the Ed25519 library. Some other API techniques can be found in section 4.2.

In TLS 1.3, the two-party computation on key scheduling becomes the bottleneck of the entire protocol. In [27], it is estimated that the 2PC circuit involves roughly 30 invocations of 2PC-SHA256 (around 75.6s using the running time over WAN in [27]). In this paper, we investigate how to efficiently design two different types of 2PC-HMAC: shared message and shared key ([27] only works on 2PC-HMAC with shared message for TLS 1.2). We also design our own optimized 2PC for modular addition over the finite field of Curve25519. We saved around 50% of AND gates as compared to the traditional method. Details can be found in section 5.

We relax the *privacy* requirement from DECO, so that we only consider the privacy of the prover's input only. As a result, a number of invocations to 2PC-HMAC and 2PC-AES are saved.

Finally, we apply our techniques and build our solution for decentralized oracles over TLS 1.3 as Decentralized IDentification Oracles (DIDO). We demonstrate that DIDO can access restricted information from some utility webpages within 10 seconds.

2 Background

2.1 TLS 1.3

Transportation Layer Security (TLS) provides secure communication between web browsers and servers. Symmetric key encryption is used to encrypt the data transmitted. The keys are uniquely generated for each connection and are based on a shared secret negotiated at the beginning of the session, also known as a TLS handshake.

TLS 1.3 protocol includes a lot of security and performance improvements as compared to TLS 1.2. Two round-trips have been needed to complete the TLS 1.2 handshake. With TLS 1.3, it requires only one round-trip. TLS 1.3 removes obsolete and insecure features from TLS 1.2, such as SHA-1, RC4, DES, 3DES, AES-CBC, MD5, etc.

Key Exchange. TLS 1.3 supports Diffie-Hellman Ephemeral key exchange over finite field (DHE) and elliptic curve (ECDHE). Most servers support ECDHE because of its efficiency. TLS 1.3 currently supports ECDHE over five elliptic curves: secp256r1, secp384r1, secp521r1, Curve25519, Curve448.

TLS 1.3 defines a group generator G for each curve. In ECDHE, the client picks a random point xG and the server picks a random point yG. The secret generated after the key exchange is the *x-coordinate* of xyG.

2.2 Two-Party Computation

Two-Party Computation (2PC) allows two parties to jointly compute an arbitrary function on their inputs without sharing the value of their inputs with the other party.

There are two major types of 2PC protocols. Garbled-circuit protocols based on Yao [24] can be used for 2PC over symmetric key cryptosystem such as AES or SHA256. For arithmetic operations, one can design a specific 2PC protocol to achieve better efficiency. For example, the Multiplicative-to-additive (MtA) [10] allows Alice (holding a secret α) and Bob (holding a secret β) to obtains shares x and y respectively, such that $\alpha\beta = x + y$.

2.3 Zero-knowledge Proof

Zero-knowledge proof is a protocol allowing a prover P to convince a verifier V that a statement is true without disclosing any other information. We mainly consider zero-knowledge proof for NP language here. Denote R a polynomial time decidable binary relation and L_R the NP language: $L_R = \{x | \exists w \text{ s.t. } (x, w) \in R\}$ where w is the witness for statement x. In this paper, we consider the classical 3-move Σ-protocol, including commit, challenge and response, between P and V. This protocol satisfies the following three properties:

1. **Completeness**: With auxiliary input w, the prover can convince the verifier with overwhelming probability if $x \in L_R$.
2. **Special soundness**: Given two given transcripts (a, c, z) and (a, c', z') for statement x by rewinding, and keep the same commit a for both transcripts, and the challenge c and the response z are changed. With this setting, there exists a PPT extractor which can compute witness w s.t. $(x, w) \in R$ in polynomial time with non-negligible probability.
3. **Zero-Knowledge**: For language L_R, for every PPT verifier V^*, there exists PPT simulator $\mathcal{S}$ s.t. the two ensembles $\{View_V^P(x)\}_{x \in L_R}$ and $\mathcal{S}(x)_{x \in L_R}$ are identical.

Additionally, we note that Σ protocol can be converted to non-interactive zero-knowledge (NIZK) by using Fiat-Shamir transformation in the random oracle model.

3 Data Provenance with TLS 1.3

DECO [27] allows users to prove that a piece of data accessed via TLS came from a particular website. Their framework works for both TLS 1.2 and 1.3. However, a straightforward implementation of [27] in TLS 1.3 cannot access a real-world TLS 1.3 website before time-out. In this paper, we use the DECO framework as the basic system architecture, and propose optimization for supporting the real-world TLS 1.3 website.

DECO [27] also supports (optionally) proving statements about the TLS-encrypted data in zero-knowledge. The similar technique can also be applied to our scheme. We will not further discuss the selective opening since it is out of the scope of the paper. We consider the simple case that the verifier obtains the entire message sent from the server. As a result, we modify the system requirements and security definitions of decentralized oracles in [27] for this relaxed definition.

3.1 Notations and Definitions

We denote by $\mathcal{P}$ the prover, $\mathcal{V}$ the verifier and $\mathcal{S}$ the TLS server. We use the ideal protocol execution [4] and model the essential properties (with relaxed privacy) using a functionality $\mathcal{F}$ in Fig. 4. Messages are tagged with a unique session identifier sid to separate parallel execution of $\mathcal{F}$.

$\mathcal{F}$ takes θ_s for $\mathcal{P}$ as private input, and take a query template Query and a statement Stmt for $\mathcal{V}$ as input. For example, $\theta_s =$ (username, password) is the private input of $\mathcal{P}$, and $Q = \mathsf{Query}(\theta_s)$ is the data request sent to the gas company user account webpage using the username and password in θ_s. Denote the honest response from the server by $R = \mathcal{S}(Q)$. Finally, $\mathcal{V}$ obtains some information $\mathsf{Stmt}(R)$. For example, if $\mathcal{V}$ is allowed to obtain the complete response from $\mathcal{S}$, the Stmt is an identity function; if $\mathcal{V}$ only wants to learn partial information (e.g., the 8-th to the 10-th byte of R), then $\mathsf{Stmt}(R) \subset R$.

Input. The prover $\mathcal{P}$ takes θ_s as a private input. The verifier $\mathcal{V}$ holds a query template Query and a statement Stmt. The server $\mathcal{S}$ has no input.

Functionality $\mathcal{F}$.

- At any moment during the session, for a message $(\mathsf{sid}, \mathsf{receiver}, \mathsf{m})$ in which $\mathsf{receiver} \in \{\mathcal{P}, \mathcal{V}, \mathcal{S}\}$ is received from $\mathcal{A}$, forward $(\mathsf{sid}, \mathsf{m})$ to $\mathsf{receiver}$ and forward any responses to $\mathcal{A}$.
- Upon receiving $(\mathsf{sid}, \mathsf{Query}, \mathsf{Stmt})$ as input from $\mathcal{V}$, send $(\mathsf{sid}, \mathsf{Query}, \mathsf{Stmt})$ to $\mathcal{P}$. Wait for $\mathcal{P}$ reply with message "ok" and θ_s.
- Compute $Q = \mathsf{Query}(\theta_s)$ and send (sid, Q) to $\mathcal{S}$, then record $\mathcal{S}$'s response (sid, R). Send $(\mathsf{sid}, |Q|, |R|)$ to $\mathcal{A}$.
- Send (sid, Q) to $\mathcal{P}$ and $(\mathsf{sid}, \mathsf{Stmt}(R), \mathcal{S})$ to $\mathcal{V}$.

Fig. 4: The functionality $\mathcal{F}$ of decentralized oracles.

We define the decentralizaed oracles as the protocol that does not require any server-side collaboration.

Definition 1. *A decentralized oracle protocol for TLS is a three party protocol* $\mathsf{P} = (\mathsf{P}_\mathcal{S}, \mathsf{P}_\mathcal{P}, \mathsf{P}_\mathcal{V})$ *such that* P *realizes* $\mathcal{F}$ *and* $\mathsf{P}_\mathcal{S}$ *is the standard TLS with an application-layer protocol.*

Adversarial Model. We consider a static network adversary $\mathcal{A}$. There are two possible models for corrupting $\mathcal{P}$ or $\mathcal{V}$.

- Semi-honest: $\mathcal{P}$ or $\mathcal{V}$ may reveal their states to $\mathcal{A}$, but it still follows the protocol.
- Malicious: $\mathcal{P}$ or $\mathcal{V}$ may deviate arbitrarily from the protocol and reveal their states to $\mathcal{A}$.

Security Properties. The security holds when either $\mathcal{P}$ or $\mathcal{V}$ is corrupted. There are two security guarantees for the functionality $\mathcal{F}$ in the malicious adversarial model.

- Prover-integrity: An adversarial $\mathcal{P}$ cannot cause $\mathcal{S}$ to accept invalid queries, or incorrectly answer to valid ones, nor she forges content provenance. If $\mathcal{V}$ outputs $(\mathsf{Stmt}(R), \mathcal{S})$ with the input $(\mathsf{Query}, \mathsf{Stmt})$, we must have $R = \mathcal{S}(Q)$ after $\mathcal{P}$ sending $Q = \mathsf{Query}(\theta_s)$ to $\mathcal{S}$ in a TLS session.
- Privacy: An adversarial $\mathcal{V}$ only learns public information $(\mathsf{Query}, \mathcal{S})$ and obtains $\mathsf{Stmt}(R)$.

We do not consider verifier integrity as in [27]. It is because the prover who has the private input θ_s can always make a query to the server by himself and obtains $R = \mathcal{S}(Q)$. Here, we assume that the server will not reply a randomized response. Hence, we do not need verifier-integrity under this assumption.

In this paper, we can achieve security against malicious adversary if the underlying building blocks are also secure against malicious adversary. However due to the performance issue (to be discussed in the next section), we can only select building blocks secure against semi-honest adversary in our implementation. Hence, the final implementation is secure against semi-honest adversary in this case. If we only consider semi-honest adversary, prover-integrity follows immediately since the prover will not deviate from the protocol.

3.2 Estimating the Performance of DECO with TLS 1.3

DECO [27] mainly implemented building blocks for TLS 1.2 and obtained some online and offline running time in the WAN setting. The source code of DECO is not publicly available. We perform a rough estimation on implementing DECO with TLS 1.3 and see if it will trigger a timeout.

Running time of ECtF. [27] only provided the running time of 2.85s (online) and 10.29s (offline) for running the three-party handshake, which mainly consists of an iteration of ECtF and also a 2PC-SHA256 for deriving the master secret. The running time of the 2PC-SHA256 is similar to the running time of 2PC-HMAC in [27], which is about 2.52s (online) and 3.19s (offline). It implies that the running time of ECtF is about 0.33s (online) and 7.1s (offline).

Running time of TLS 1.3 key scheduling. As estimated in [27], the 2PC circuit for TLS 1.3 key scheduling roughly takes 30 invocations of 2PC-SHA256. It takes 75.6s (online) and 95.7s (offline).

Running time of query execution. As shown in [27], the running time of 2PC-AES-GCM for 256 bytes data is 1.21s (online) and 12.01s (offline).

We can see the estimated running time in the WAN setting is 77.14s (online) and 114.81s (offline), even without doing any selective opening in [27]. It is very likely that it will trigger a timeout and hence it is not practical.

3.3 Overview of Our Design

There is a huge gap between the estimated running time of 77.14s and our target running time of 10s. The most significant part (98%) of the running time comes from the large number of 2PC-SHA256 in TLS 1.3 key-scheduling.

The TLS 1.3 key scheduling is illustrated in Fig. 2 of [8]. In our Fig. 5, we extend it by including the final session keys used for encrypting the application traffic ($\mathsf{tk}_{\mathsf{capp}}$ for client's key and $\mathsf{tk}_{\mathsf{sapp}}$ for server's key), which are derived from CATS and SATS. Assume that we do not have pre-shared key (PSK) for the initial connection. We can treat the keys derived only from PSK as constant. Hence we can simplify the key scheduling by considering the key dES as a constant number. The other input DHE is the output of the ECtF protocol, which is shared by the prover and the verifier. As estimated in DECO [27], each 2PC computation of HKDF.Extract and HKDF.Expand requires two or three invocations of 2PC-SHA256 (depending on which input is shared, refer to section 5.1 and 5.2 for details). Since there are 15 invocations of HKDF.Extract and HKDF.Expand, it sums up to at least 44 invocations of 2PC-SHA256. To illustrate the complexity of TLS 1.3 key scheduling, the key derivation of TLS 1.2 is shown in Fig. 6 using similar notation.

Design 1 for Selective Opening. We first design a 2PC key scheduling to minimize the number of invocations of 2PC-SHA256. In order to support selective opening in [27], the server's application

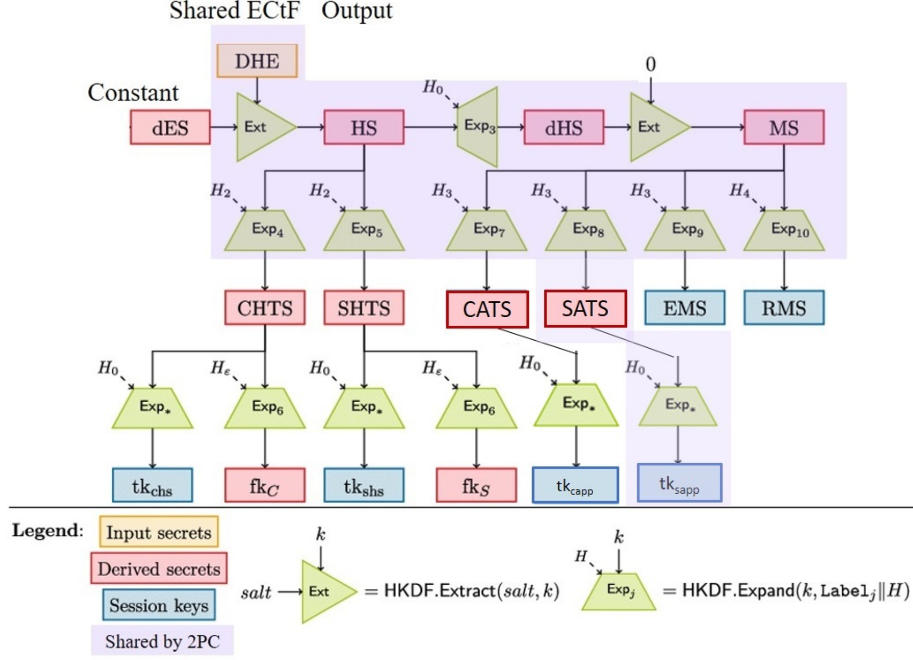

Fig. 5: Design 1 for 2PC key scheduling for TLS 1.3. The definition of strings $\texttt{Label}_j$ and H_i in TLS 1.3 can be found in [8].

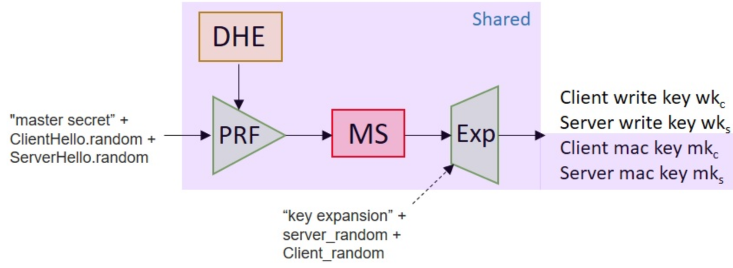

Fig. 6: 2PC key derivation for TLS 1.2.

key $\mathsf{tk}_{\mathsf{sapp}}$ should be shared between the prover and the verifier. As a result, the secrets between DHE and $\mathsf{tk}_{\mathsf{sapp}}$ should be shared and computed by 2PC. On the other hand, the secrets used to derive the handshake keys ($\mathsf{CHTS}, \mathsf{SHTS}$) and the client's application key (CATS) can be obtained by the prover. As shown in the purple area in Fig. 5, we save five 2PC computations of $\mathsf{HKDF.Expand}$, which is equivalent to 15 invocations of 2PC-SHA256.

Design 2 without Selective Opening. Although our first design already reduces 1/3 running time in key scheduling, we still want to further optimize in TLS 1.3. Assume that selective opening is *not* needed in the application. Note that SNARK is only needed for achieving selective opening. Therefore, the implementation of SNARK is out of the scope of this design. Recall that the server's application key $\mathsf{tk}_{\mathsf{sapp}}$ is used to decrypt and to authenticate the data sent from the server. Hence, it should not be fully revealed to the prover during key scheduling. In order to reduce the number of 2PC computation, we set $\mathsf{tk}_{\mathsf{sapp}}$ and SATS to be completely revealed to the verifier.

Since the client's application key $\mathsf{tk}_{\mathsf{capp}}$ is used to encrypt some secret information of the prover (e.g., password or cookie file), $\mathsf{tk}_{\mathsf{capp}}$ and CATS should not be known to the verifier. It implies that the key MS should be shared between the prover and the verifier. The overall design for the 2PC key scheduling is shown in Fig. 7.

This design has three improvements in terms of running time, when selective opening is not needed:

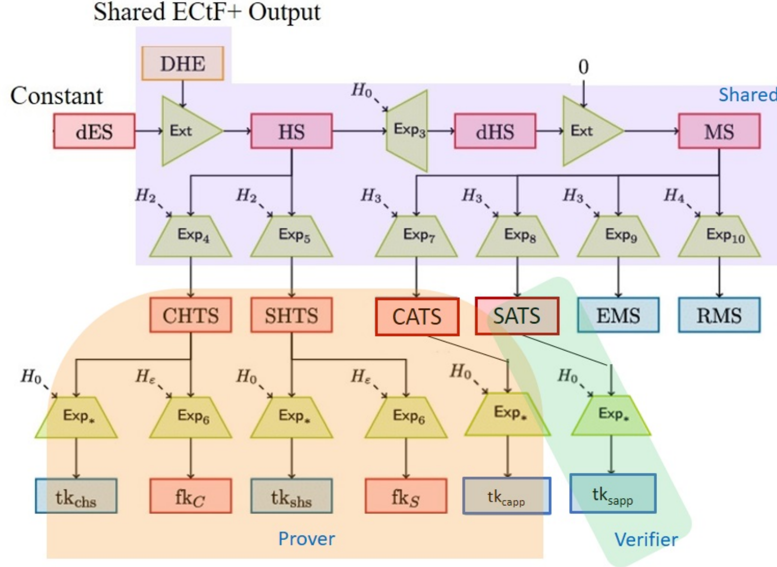

Fig. 7: Design 2 for 2PC key scheduling for TLS 1.3 (without selective opening).

1. Reduce three invocations of 2PC-SHA256 in key scheduling, as compared to our first design in Fig. 5.
2. Remove the 2PC-AES-GCM for query execution. The verifier can decrypt the HTML page completely by himself.
3. Save the running time for selective opening.

In the rest of the paper, we will implement our **second design** for efficiency reason. Note that our first design can still be used if the underlying 2PC-SHA256 and 2PC-AES-GCM is fast enough for completing the TLS 1.3 connection. However, our DIDO is designed in a semi-honest setting due to the performance bottleneck of 2PC protocols. Please refer to Appendix B for further discussion about the performance bottleneck among the existing 2PC protocols.

Based on our design 2 and the TLS 1.3 protocol, the overall system architecture for our scheme is modified, as shown in Fig. 3.

4 Three-party ECDHE

In DECO [27], the prover and the verifier jointly act as the client and interact with the server. Their three-party handshake (3P-HS) involves a ECDHE key exchange, the ClientHello and ServerHello messages in TLS 1.2, and the key derivation function. In this paper, we will separate them and analyse each protocol individually.

The three-party ECDHE (3P-DH) runs as follows.

1. The verifier picks a random x_v and sends x_vG to the prover. The verifier computes a zero-knowledge proof π_v of x_v with respect to x_vG.
2. If π_v passes the verification, the prover picks a random x_p and sends $x_pG + x_vG = (x_p + x_v)G$ to the server following the TLS 1.3 ECDHE protocol.
3. The server replies with yG. The prover forwards it to the verifier. The prover also sends to the verifier x_pG and a zero-knowledge proof π_p of x_p with respect to x_pG. The prover computes x_pyG and the verifier computes x_vyG if π_p passes the verification.
4. The prover and the verifier have to perform a two-party computation for the generated session key, i.e., sharing the x-coordinate of $(x_p + x_v)yG = x_pyG + x_vyG$. This two-party computation is called ECtF: converting shares in $EC(\mathtt{F}_p)$ to shares in $\mathtt{F}_p$.

DECO [27] used the secret-sharing-based Multiplicative-to-Additive (MtA) protocols in [10] to construct ECtF. There are two main issues for ECtF in DECO. Firstly, DECO is expensive in

terms of round complexity. DECO used 8 rounds of communication, which includes 6 rounds for using the 3 MtA protocols in ECtF. The online running time of DECO's handshake protocol in the LAN and WAN is 368.5ms and 2850ms respectively [27]. The difference is mainly caused by the round-trip time 67ms between two nodes in their WAN setting. In other words, the 8 rounds of communication in ECtF already used 9.4% (268ms) of the running time. The second issue is that DECO only considers ECDHE on the elliptic curve of the form $Y^2 = X^3 + a_1X + a_0 \bmod p$ (e.g., secp256r1, secp384r1 and secp521r1 in TLS 1.3). Their ECtF does not support the Montgomery curve of the form $Y^2 = X^3 + a_2X^2 + a_1X \bmod p$ (e.g., Curve25519 and Curve448 in TLS 1.3), which is also widely used in practice.

4.1 Round-optimal ECtF+ Protocol for All TLS 1.3 Curves

In this paper, we propose an improved version of the ECtF protocol, which supports all elliptic curves in the TLS 1.3 standard. Instead of using the MtA protocols in [10] in a black-box manner, we design our two-party computation protocol from scratch and construct a round-optimal $\mathsf{ECtF}+$ protocol. It only has 3 rounds of communication (even ECDHE has 2 rounds: Alice sends an ephemeral key to Bob and Bob sends an ephemeral key to Alice). In addition, it supports all elliptic curves in the TLS 1.3 standard.

Point Addition on Elliptic Curve. Consider an elliptic curve of the general form:

$$v^2 = u^3 + a_2 \cdot u^2 + a_1 \cdot u + a_0 \mod p.$$

For Curve25519, $a_2 = 486662, a_1 = 1, a_0 = 0$ and $p = 2^{255} - 19$. For secp256r1, $a_2 = 0, a_1 = -3$, $a_0 = 410583637251521\ 421293261297800472684091144410159937255548352563\ 14039467401291$ and $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

Consider that the prover P and the verifier V have ECC points $P_1 = (u_1, v_1)$ and $P_2 = (u_2, v_2)$ respectively. They want to jointly compute $K = P_1 + P_2$ and they get k_1 and k_2 such that $k_1 + k_2 = u_k \bmod p$ and $K = (u_k, v_k)$. Recall that by the elliptic curve computation, we have $u_k = \lambda^2 - a_2 - u_1 - u_2$, where $\lambda = (v_2 - v_1)/(u_2 - u_1)$.

Our Scheme. The intuition of our $\mathsf{ECtF}+$ protocol is to run two specially designed MtA protocols in parallel. Suppose that the prover P chooses a randomness r_1 and the verifier V chooses a randomness r_2. P and V run the MtA for $\delta = (r_1 + r_2)(u_1 - u_2)$ and $\omega = (r_1 + r_2)(v_1 - v_2)$ in parallel. Hence they can obtain $\lambda = \delta/\omega$ and calculate $u_k = \lambda^2 - a_2 - u_1 - u_2$ accordingly.

Our $\mathsf{ECtF}+$ protocol is as follows.

[**Offline**] Suppose that $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ is an additive homomorphic encryption scheme. P runs $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}()$ and sends pk to the verifier V.

[**Online**] The prover P and the verifier V has a private input (u_1, v_1, sk) and (u_2, v_2) respectively.

1. P picks a random $r_1 \in \mathbb{Z}_p$ and computes:

$$C_u := \mathsf{Enc}_{\mathsf{pk}}(u_1), \quad C_v := \mathsf{Enc}_{\mathsf{pk}}(v_1), \quad C_r := \mathsf{Enc}_{\mathsf{pk}}(r_1), \quad C_{rv} := \mathsf{Enc}_{\mathsf{pk}}(r_1 v_1).$$

   P sends (C_u, C_v, C_r, C_{rv}) to V.
2. V picks some random $r_2, \beta_2, \gamma_2, \alpha_2 \in_R \mathbb{Z}_p$ and computes:

$$\begin{aligned}
C_\beta &:= C_u^{r_2} \cdot C_r^{-u_2} \cdot \mathsf{Enc}_{\mathsf{pk}}(-\beta_2) = \mathsf{Enc}_{\mathsf{pk}}(u_1 r_2 - u_2 r_1 - \beta_2),\\
C_\gamma &:= C_v^{r_2} \cdot C_r^{-v_2} \cdot \mathsf{Enc}_{\mathsf{pk}}(-\gamma_2) = \mathsf{Enc}_{\mathsf{pk}}(v_1 r_2 - v_2 r_1 - \gamma_2),\\
\delta_2 &:= \beta_2 - r_2 u_2 \mod p,\\
\omega_2 &:= \gamma_2 - r_2 v_2 \mod p,\\
C_\alpha &:= (C_\gamma \cdot C_{rv})^{\omega_2} \cdot \mathsf{Enc}_{\mathsf{pk}}(-\alpha_2).
\end{aligned}$$

   V sends $(C_\beta, C_\gamma, \delta_2, C_\alpha)$ to P.
3. P runs $\beta_1 := \mathsf{Dec}_{\mathsf{sk}}(C_\beta)$, such that $\beta_1 + \beta_2 = u_1 r_2 - u_2 r_1$. P first sends $\delta_1 := \beta_1 + r_1 x_1 \bmod p$ to V.

P locally runs $\gamma_1 := \mathsf{Dec}_{\mathsf{sk}}(C_\gamma)$, such that $\gamma_1 + \gamma_2 = v_1 r_2 - v_2 r_1$. P computes:

$$\begin{aligned}
&\delta := \delta_1 + \delta_2 = u_1 r_2 - u_2 r_1 + r_1 u_1 - r_2 u_2 = (r_1 + r_2)(u_1 - u_2) \mod p,\\
&\omega_1 := \gamma_1 + r_1 v_1 \mod p,\\
&\alpha_1 := \mathsf{Dec}_{\mathsf{sk}}(C_\alpha) = (\gamma_1 + r_1 v_1)\omega_2 - \alpha_2,\\
&s_1 = (\omega_1^2 + 2\alpha_1)\delta^{-2} - a_2 - u_1 \mod p.
\end{aligned}$$

P outputs s_1.

4. V computes:

$$\begin{aligned}
\delta &:= \delta_1 + \delta_2 \mod p,\\
s_2 &:= (\omega_2^2 + 2\alpha_2)\delta^{-2} - u_2 \mod p.
\end{aligned}$$

V outputs s_2.

Correctness. We can check that s_1 and s_2 are the additive shares of u_k.

$$\begin{aligned}
\omega_1 + \omega_2 &= (\gamma_1 + \gamma_2 + r_1 v_1 - r_2 v_2)\\
&= (v_1 r_2 - v_2 r_1 + r_1 v_1 - r_2 v_2)\\
&= (r_1 + r_2)(v_1 - v_2)\\
&= \delta(v_1 - v_2)/(u_1 - u_2) = \delta\lambda.\\
\alpha_1 + \alpha_2 &= (\gamma_1 + r_1 v_1)\omega_2 = \omega_1\omega_2.\\
s_1 + s_2 &= (\omega_1^2 + 2\alpha_1)\delta^{-2} - a_2 - u_1 + (\omega_2^2 + 2\alpha_2)\delta^{-2} - u_2\\
&= (\omega_1^2 + 2(\alpha_1 + \alpha_2) + \omega_2^2)\delta^{-2} - a_2 - u_1 - u_2\\
&= (\omega_1 + \omega_2)^2\delta^{-2} - a_2 - u_1 - u_2\\
&= \lambda^2 - a_2 - u_1 - u_2.
\end{aligned}$$

Simulation. If the adversary corrupts P, then V's message can be simulated without knowledge of its input (u_2, v_2). Indeed a simulator can just choose a random $(u_2', v_2') \in \mathbb{Z}_p^2$ and act as V. The distribution of the message decrypted by P in this simulation is statistically close to the message decrypted when V uses the real (u_2, v_2), because the "noise" $\beta_2, \gamma_2, \alpha_2$ are uniformly distributed in $\mathbb{Z}_p$.

If the adversary corrupts V, then P's message can be simulated without knowledge of its input (u_1, v_1, sk). Indeed a simulator can just choose a random a $(u_1', v_1') \in \mathbb{Z}_p^2$ and act as Alice. In this case the view of V is computationally indistinguishable from the real one due to the semantic security of the encryption scheme and the random choice of r_1 in $\mathbb{Z}_p$.

Security in the Malicious Setting. The protocol mentioned above is secure in the semi-honest setting. If we want to make it secure in the malicious setting, the online protocol has to be modified as follows.

1. P additionally outputs a non-interactive zero-knowledge (NIZK) proof π_P for (u_1, v_1, r_1) with respect to (C_u, C_v, C_r, C_{rv}).
2. V verifies if π_P is valid. V additionally outputs a NIZK proof π_V for $(r_2, \beta_2, \gamma_2, \alpha_2)$ with respect to $(C_\beta, C_\gamma, C_\alpha)$.
3. P verifies if π_V is valid.

A detailed summary of NIZK proofs used in the MtA protocol was given in [23]. If the underlying encryption scheme is the Paillier encryption [16], then the prover P needs to provide an additional NIZK proof of the well-formedness of the Paillier public key N in the offline phase. In the online phase, π_V can be omitted by using the Paillier-EC assumption [10]. For each Paillier ciphertext in step 1, the corresponding NIZK proof takes 6 Paillier exponentiations and sends extra 6 log N bits. We emphasize that in this paper, we will not adopt the malicious setting of $\mathsf{ECtF}+$ since our DIDO is designed in a semi-honest setting due to the performance bottleneck of 2PC protocols. Therefore, we further relax other parts of DIDO to the semi-honest model for optimization, including the

Table 1: Online running time of $\mathsf{ECtF}(+)$ on different elliptic curves in Paillier and CL encryption, in the LAN setting.

	Paillier		CL	
Curves	ECtF [27]	Our ECtF+	ECtF [27]	Our ECtF+
secp256r1	0.350s	0.308s	3.012s	2.544s
Curve25519	× (0.363s)	0.317s	× (2.971s)	2.660s

Table 2: Online running time of $\mathsf{ECtF}(+)$ on different elliptic curves in Paillier encryption, in the WAN setting.

	Paillier	
Curves	ECtF [27]	Our ECtF+
secp256r1	0.423s	0.386s
Curve25519	× (0.428s)	0.386s

removal of zero-knowledge proofs for MtA. Please refer to Appendix B for further discussion about the performance bottleneck among the existing 2PC protocols.

ECtF+ Implementation. We implement DECO's ECtF and our ECtF+ and run them in both the LAN and the WAN setting for direct comparison. We test the schemes by using both the Paillier encryption and the CL encryption [5]. We use the *rust-paillier* library for the Paillier encryption and the *class* library for the CL encryption. We use both the curve secp256r1 and Curve25519 in the implementation. In the LAN setting, the prover and the verifier are run in MacBook Pro (resp. 3.1GHz dual-core Intel Core i5 and 1.4GHz quad-core Intel Core i7). They are put under the same WiFi. For the online part, the prover's running time is similar to the verifier's running time. We take the maximum running time of the prover and the verifier for simplicity. The results are shown in Table 1. We also modify ECtF to support Curve25519 for comparison (the running time is shown in brackets in the tables).

In the WAN setting, the prover and the verifier are run in two Azure virtual machines with 2.4GHz Intel Xeon CPU. They are in two different locations in a country. The round-trip time in the WAN setting is 58 ms. Since we observed that the Paillier version is 10 times more efficient than the CL version. Hence we only proceed to the Paillier version in the WAN setting. The results are shown in Table 2.

Our ECtF+ outperforms DECO's ECtF in three ways. Firstly, our ECtF+ is around 7.9% faster than DECO's ECtF if it is run in the LAN setting, as shown in Table 1. Secondly, our scheme has only 3 rounds of communication while [27] has 8 rounds. Our ECtF+ is around 13.1% faster than DECO's ECtF if it is run in the WAN setting, as shown in Table 2. Thirdly, our ECtF+ supports the efficient operation over Curve25519. We choose to use X25519 for the TLS handshake protocol for the rest of the paper.

4.2 Three-party ECDHE with X25519

In TLS 1.3 ECDHE [17], the elliptic curve points are sent differently for different curves. For the curves secp256r1, secp384r1, and secp521r1, the binary representation of the entire (x, y)-coordinate is sent. However, only the u-coordinate on the Montgomery curve is sent for X25519 and x448 as shown in RFC 7748 [14]. In TLS libraries that support X25519 and X448, the API usually outputs the u-coordinate for ECDHE computation only. It is not compatible with the ECtF protocol, since we need the (u, v)-coordinates of $x_p yG$ and $x_v yG$ for the prover and the verifier respectively as the input to ECtF.

The X25519 standard is defined in RFC 7748. They define a function X25519(k, u), where k is a 32 bytes string and u is a u-coordinate. The function first decodes k as an integer scalar, sets the three least significant bits of the first byte and the most significant bit of the last to zero, sets the second most significant bit of the last byte to 1 and, finally, decodes as little-endian. The scalar multiplication can be computed by the decoded number and u, using the Montgomery formula (a pseudocode is given in RFC 7748). If Alice and Bob choose random strings k_a and k_b

respectively, and u_* is the base point of Curve25519, their session key is X25519(k_a, X25519(k_b, u_*)) = X25519(k_b, X25519(k_a, u_*)). The case of X448 is almost the same and we omit it for simplicity.

The integration of the three-party handshake (3P-HS) protocol in [27] with X25519 is not straightforward. There are a few reasons:

1. The function X25519($\cdot, \cdot$) defined in RFC 7748 only returns the u-coordinate. With only the u-coordinate, the prover cannot calculate point addition (Step 2 of 3P-DH), and also cannot run the ECtF protocol (Step 4). We need to have the (u, v)-coordinate.
2. There is a lack of Curve25519 point addition API. The only compulsory API is X25519($\cdot, \cdot$) as defined in RFC 7748. Some TLS libraries do not provide Curve25519 point addition API.
3. The scalar k is masked before use.

In order to support X25519 and X448, one needs to find a TLS library without the above obstacles, or develop his own library (which is time consuming and error-prone). In this paper, we propose an alternative approach.

Workaround for v-coordinate. Recall that for Curve25519, we have $v^2 = u^3 + 486662u^2 + u \mod p$. Given the u-coordinate, we can calculate v by taking the square root modulo p, using the Tonelli-Shanks algorithm. However, there are two possible values: $\pm v \mod p$. We will discuss how to handle this problem later.

Workaround for point addition: API from EdDSA. We observe that EdDSA signatures (Ed25519 and Ed448) are also included in TLS 1.3. The twisted Edwards curve $-x^2+y^2 = 1+dx^2y^2 \mod p$ (for some constant d) is equivalent to the Montgomery curve. API and pseudocode for point addition over the twisted Edwards curve are provided in RFC 8032 [11]. Once we have a (u, v)-coordinate in the Montgomery curve, we convert it to the coordinates in the twisted Edwards curve. Then we call the point addition function over the twisted Edwards curve, and convert it back to the (u, v)-coordinate in the Montgomery curve. Hence we can handle point addition using the existing TLS 1.3 API for EdDSA.

In our implementation, the Rustls library uses the ring library [19] for point addition over the twisted Edwards curve. The points are stored in a projective coordinate (X, Y, Z) satisfying $x = X/Z, y = Y/Z$. Given a (u, v)-coordinate on Curve25519, it is converted to a projective coordinate $(X, Y, 1)$, where:

$$X = \sqrt{-486664} \cdot \frac{u}{v}, \quad Y = \frac{u-1}{u+1}.$$

After working on point addition over the (X, Y, Z)-coordinate, we eventually convert it back to the (u, v)-coordinate by:

$$(u, v) = (\frac{Z+Y}{Z-Y}, \quad \sqrt{-486664} \cdot \frac{uZ}{X}).$$

Workaround for $\pm v$ and masked k. Assume that the u-coordinate of the generator G is u_G. Observe that in step 1 of 3P-HS with X25519, the verifier picks a random k_v and runs X25519(k_v, u_G). The input k_v is masked to x_v, and then the u-coordinate of x_vG is returned and is sent to the prover. In step 2, the prover computes the v-coordinate of either x_vG or $-x_vG$ because there are two possible values. The u-coordinate of $(x_p \pm x_v)G$ is sent to the server. In step 3, the prover receives a u-coordinate u_s from the server. The prover uses X25519(k_p, u_s) to obtain the u-coordinate and calculates the v-coordinate of x_pyG. Similarly, the verifier computes x_vyG. In step 4, the ECtF protocol outputs the shares of the u-coordinate of $(x_p + x_v)yG$ or $(x_p - x_v)yG$. It is the same as the session key computed by the server with 50% probability.

We propose a solution for this problem. After running X25519(k_v, u_G), we can run the masking function on k_v to obtain x_v, use the scalar multiplication API from EdDSA to calculate x_vG, convert back to the Montgomery curve, and check whether the v-coordinate is negative. If so, change x_v to $-x_v$ (to flip the sign of the v-coordinate) and find the corresponding masked k'_v. It can ensure that point addition is always performed in step 2. However, this k'_v may not exist since it has to follow a specific masking format. Hence, we need to pick another k_v until the v-coordinate is positive.

5 Design for 2PC Key Scheduling

In this section, we will discuss how to implement the 2PC key scheduling design in Fig. 3.

2PC Key Scheduling for TLS 1.3. Recall that after running $\mathsf{ECtF+}$, the prover and the verifier have the additive shares of the ECDHE session key DHE. The goal of our key scheduling is to ensure that the prover cannot obtain the server application traffic secret SATS, which is used to authenticate the information returned by the server. On the other hand, we also do not want the verifier to obtain the client application traffic secret CATS, since it is used to encrypt the information sent from the prover. As a result, we need to set MS as the shared secret between the prover and the verifier according to Fig. 3. It further implies that dHS and HS are both shared secrets. For the generated session keys, it is safe to give the client handshake key $\mathsf{tk_{chs}}$ and the server handshake key $\mathsf{tk_{shs}}$ to the prover. Hence, we need to use 2PC computation for the purple box in 3.

In order to complete the 2PC key scheduling with shared ECDHE session key DHE as input, we need to implement 2PC for the functions $\mathsf{HKDF.Extract}$ and $\mathsf{HKDF.Expand}$. In TLS 1.3, we have

$$\mathsf{HKDF.Extract}(salt, k) = \mathrm{HMAC}(salt, k),$$
$$\mathsf{HKDF.Expand}(k, \mathtt{Label}_j || H_i) = \mathrm{HMAC}(k, \mathtt{Label}_j || H_i).$$

The definition of the constant $\mathtt{Label}_j$ and H_i can be found in [8]. Recall the definition of HMAC function for a key K and a message M (when $|K|$ matches the key length of the hash function H)

$$\mathrm{HMAC}(K, M) = H((K \oplus opad) || H((K \oplus ipad) || M)),$$

where $opad$ is 512 bits of repeated bytes 0x5c, $ipad$ is 512 bits of repeated bytes 0x36. From Fig. 3, we need to use both 2PC-HMAC for shared message M and 2PC-HMAC for shared key K.

5.1 2PC-HMAC for Shared Message

We only need to use 2PC-HMAC for shared message once, where the message $M = \mathsf{DHE}$ is shared by the prover and the verifier and the key $K = \mathsf{dES}$ is a constant. We use SHA-256 as the hash function H in our implementation, which is supported by TLS 1.3. If we break down HMAC by the SHA-256 compression function $\mathsf{SHA256}(\cdot, \cdot)$, we can see that the computation of $\mathrm{HMAC}(\mathsf{dES}, \mathsf{DHE})$ is as follows.

1. Compute the chaining state $cs_1 = \mathsf{SHA256}(IV, \mathsf{dES} \oplus ipad)$, where IV is the initialization vector for SHA-256.
2. Compute the chaining state $cs_2 = \mathsf{SHA256}(IV, \mathsf{dES} \oplus opad)$.
3. Compute the 256 bit padding pad[3]. Compute $h_1 = \mathsf{SHA256}(cs_1, \mathsf{DHE}||pad)$.
4. Output $\mathsf{HS} = \mathsf{SHA256}(cs_2, h_1||pad)$.

Since dES is a constant (when there is no pre-shared key), we can precompute cs_1 and cs_2. Recall that the prover has a share u_p and the verifier has a share u_v such that $u_p + u_x = \mathsf{DHE} \bmod p$. To compute 2PC-HMAC($\mathsf{dES}, \mathsf{DHE}$), we need to run as follows.

1. The prover and the verifier run a 2PC-Modular-Addition with private input u_p and u_v respectively. They will obtain a XOR share of DHE.
2. The prover and the verifier run a 2PC-$\mathsf{SHA256}(cs_1, \mathsf{DHE}||pad)$ with the XOR shares of DHE as input. They will obtain a XOR share of h_1.
3. The prover and the verifier run a 2PC-$\mathsf{SHA256}(cs_2, h_1||pad)$ with the XOR shares of h_1 as input. They will obtain a XOR share of HS.

We use the efficient implementation of 2PC-$\mathsf{SHA256}$ in the emp-sh2pc library [20] [4], which uses garbled circuits and oblivious transfer. Hence, we only need to design an efficient 2PC-Modular-Addition protocol.

---

[3] According to the padding rules in HMAC, the first bit is one, followed by 191 zeros, and the last 64 bits is the binary of 768.

[4] https://github.com/emp-toolkit/emp-sh2pc

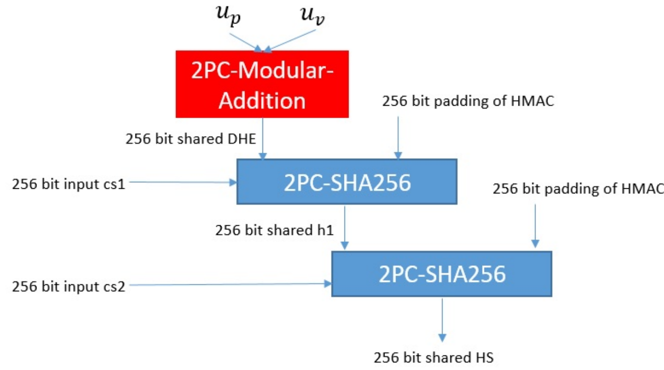

Fig. 8: Our design for 2PC-HMAC($\mathsf{dES}, \mathsf{DHE}$) with shared DHE.

2PC-Modular-Addition. There are a number of circuits designed for 2PC addition of 64 bits. To the best of the authors' knowledge, there are no modular addition circuits available for the 2PC library emp we used. Hence we design our own modular addition circuit for 2PC.

Adder. It is well-known to build a half adder with one XOR gate and one AND gate. A full adder can be constructed from two XOR gates, two AND gates and one OR gate (In Fig. 9, the last OR gate in the full adder can be replaced by one XOR gate without altering the resulting logic). To build a k-bit ripple adder with carry, we need $k-1$ full adder and 1 half adder in the least significant bit. There are some adders (e.g., Kogge-Stone, Han-Carlson, Brent-Kung) designed to reduce the depth of the circuit from k to $O(\log k)$, at a cost of using more gates.

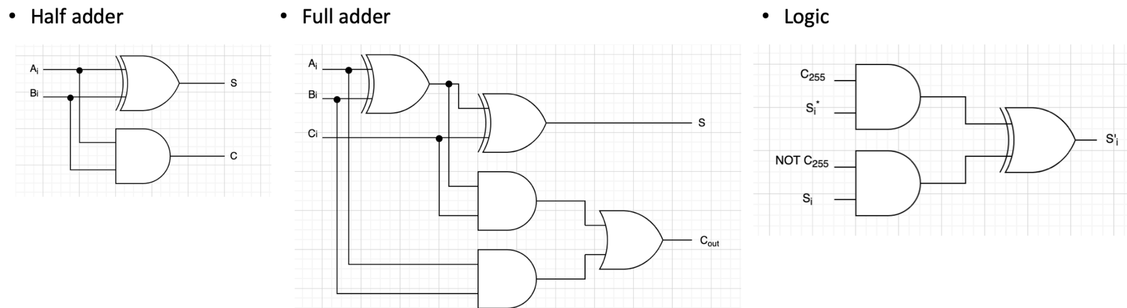

Fig. 9: Traditional half adder, full adder and "if-else" logic circuit.

The 2PC usually works for circuit with XOR, AND and INV gates. The performance of 2PC is mainly dominated by the number of AND gates in the circuit, instead of the depth of the circuit. In order to optimize the adder circuit, Goldfeder designed a full adder circuit with 1 AND gate and 4 XOR gate[5] (Fig. 10).

Our design. The design of our 2PC-Modular-Addition circuit for Curve25519 is as follows.

1. We first build a 255 bits ripple adder circuit with 254 Goldfeder's full adder and 1 half adder in the least significant bit.
2. If the final carry bit c_{255} is 1, add 19 to the 255-bit output, since $X = X–(2^{255}–19) \bmod 2^{255}–19$. Else, simply output X.

To implement the second step, we need a circuit to generate the constant term 19 and the logic gates for "if-else" logic. We observe that b XOR $b = 0$ for any $b = 0/1$. Hence the constant 0 can be generated by one XOR gate. The constant 1 can be obtained by applying an INV gate to 0. As a result, we can obtain 19 in binary (Fig. 10).

[5] http://stevengoldfeder.com/projects/circuits/sha2circuit.html

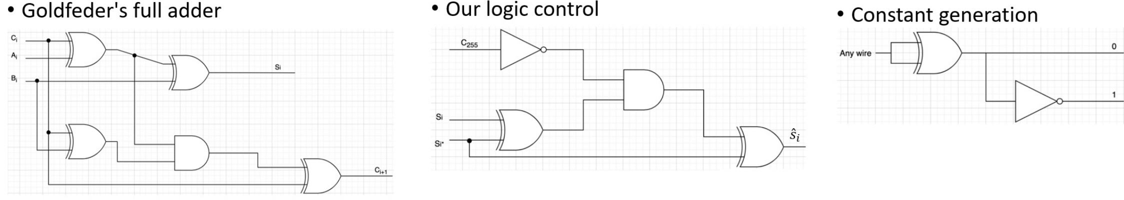

Fig. 10: Our circuit design.

Table 3: The number of logic gates used for a modular addition circuit (mod $2^{255} - 19$).

	# AND	# XOR	# INV
Traditional	1528	1783	2
This paper	765	2546	2

Logic control. The "if-else" logic can be implemented as follows. Assume that s_i is the i-th bit of a 255 bit number S, and s_i^* is the i-th bit of a 255 bit number $S + 19$. As shown in Fig. 9, the traditional "if-else" logic can be implemented by:

$$\hat{s}_i = (c_{255} \text{ AND } s_i^*) \text{ XOR } ((\text{INV } c_{255}) \text{ AND } s_i).$$

This circuit runs for 255 times and the final output is $(\hat{s}_{255}, \ldots, \hat{s}_1)$. However, this "if-else" logic circuit is not optimized for 2PC with the use of two AND gates. We give a new design by using one AND gate, one INV gate and two XOR gates. It is illustrated in Fig. 10.

Result. As compared with the traditional approach (simple full adder, half adder and logic control) in Table 3, our current approach saves about half of the AND gates. As a result, the efficiency of our 2PC-Modular-Addition can be improved by about 50%, by using the *free XOR* technique in 2PC [13].

Remark that our circuit determines the modulus $2^{255} - 19$ with only the carry c_{255}. With this operation, it leaves $2^{255} - 18$ to $2^{255} - 1$ which we treat them as the case "no need to take mod". This leads to a negligible probability of error.

5.2 2PC-HMAC for Shared Key

We need to use 2PC-HMAC for computing $\mathsf{MS} = \text{HMAC}(\mathsf{dHS}, 0)$, and all $\mathsf{HKDF.Expand}(\cdot, \cdot)$ functions with shared keys HS or MS. The computation of the 2PC-HMAC(K, M) function is as follows.

1. Compute the shared chaining state $cs_1 = \mathsf{2PC\text{–}SHA256}(IV, K \oplus ipad)$.
2. Compute the shared chaining state $cs_2 = \mathsf{2PC\text{–}SHA256}(IV, K \oplus opad)$.
3. Compute the padding pad_1 and $h_1 = \mathsf{2PC\text{–}SHA256}(cs_1, M||pad_1)$[6].
4. Compute the padding pad_2. Output $\mathsf{HS} = \mathsf{2PC\text{–}SHA256}(cs_2, h_1||pad_2)$.

Alternatively, one can output the entire cs_1 to the prover and he computes h_1 on his own in step 3. The value h_1 is treated as a public input in step 4. This modification reduces the number of 2PC–SHA256 from four to three. However, it requires the library to provide an interface to modify the internal chaining state of SHA-256 to cs_1. We do not use this method in our implementation due to the complexity in engineering work.

6 Decentralized Identification Oracles (DIDO)

In this section, we integrate our techniques above and construct a decentralized oracle for TLS 1.3. We call our scheme as Decentralized IDentification Oracles (DIDO), as our application is mainly for identification purpose. We define the DIDO protocol in Figure 11.

Next, we show that the $\mathsf{P}_{\mathsf{DIDO}}$ UC-securely realizes $\mathcal{F}$, assuming a functionality $\mathcal{F}_{\mathsf{2PC}}$ for secure 2PC-HMAC, as stated in Theorem 1.

[6]For TLS 1.3, the message M here is less than 64 bytes. Hence one invocation of 2PC–SHA256 suffices.

$\mathsf{P}_{\mathcal{S}}$: Follow the standard TLS 1.3 protocol using ECDHE for key exchange.

$\mathsf{P}_{\mathcal{P}}$ and $\mathsf{P}_{\mathcal{V}}$:

- $\mathcal{P}$ receives $(\mathsf{sid}, \mathsf{Query})$ from $\mathcal{V}$, where Query is the template of query.
- $\mathcal{P}$ starts the handshake if it chooses to proceed after examining the Query.
- **(3P-DH)** $\mathcal{P}, \mathcal{V}$ execute the three-party ECDHE protocol with $\mathcal{S}$. After Step 1-2, $\mathcal{P}$ sends the client hello message and key share $(\mathsf{sid}, m_{\mathsf{chs}})$ to $\mathcal{S}$. $\mathcal{S}$ replies with the server hello message, key share and an encrypted finish message $(\mathsf{sid}, m_{\mathsf{shs}})$. By using the server key share, $\mathcal{P}, \mathcal{V}$ execute Step 3-4 of the three-party ECDHE protocol. $\mathcal{P}$ obtains a share of the ECDHE key $\mathsf{k}^{\mathsf{DHE}}_{\mathcal{P}}$ and $\mathcal{V}$ obtains another share $\mathsf{k}^{\mathsf{DHE}}_{\mathcal{V}}$, such that $\mathsf{k}^{\mathsf{DHE}}_{\mathcal{P}} + \mathsf{k}^{\mathsf{DHE}}_{\mathcal{V}} = \mathsf{k}^{\mathsf{DHE}}$, which is the ECDHE key obtained by $\mathcal{S}$.
- **(Key Scheduling)** $\mathcal{P}, \mathcal{V}$ execute the 2PC Key Scheduling with private input $\mathsf{k}^{\mathsf{DHE}}_{\mathcal{P}}$ and $\mathsf{k}^{\mathsf{DHE}}_{\mathcal{V}}$ respectively. $\mathcal{P}$ obtains the TLS 1.3 secrets CHTS, SHTS and CATS. $\mathcal{V}$ obtains the secret SATS. $\mathcal{P}$ uses SHTS to decrypt and to validate the server finish message in m_{shs}.
- **(Query)** $\mathcal{P}$ sends the client handshake finish message $(\mathsf{sid}, m_{\mathsf{cfin}})$ to $\mathcal{S}$ using keys derived from CHTS. $\mathcal{P}$ prepares the query $\mathsf{Query}(\theta_s)$ and encrypts it to client application message m_{capp} using keys derived from CATS. $\mathcal{P}$ sends $(\mathsf{sid}, m_{\mathsf{capp}})$ to $\mathcal{S}$. $\mathcal{S}$ replies with the response $(\mathsf{sid}, m_{\mathsf{sapp}})$ to $\mathcal{P}$.
- **(Open)** After receiving a response $(\mathsf{sid}, m_{\mathsf{sapp}})$ from $\mathcal{S}$, $\mathcal{P}$ forwards it to $\mathcal{V}$. $\mathcal{V}$ then decrypts m_{sapp} using keys derived from SATS and outputs the decrypted response.

Fig. 11: The protocol $\mathsf{P}_{\mathsf{DIDO}}$.

Theorem 1. *The proposed DIDO protocol* $\mathsf{P}_{\mathsf{DIDO}}$ *UC-securely realizes* $\mathcal{F}$ *in the* $\mathcal{F}_{\mathsf{2PC}}$ *world, assuming the discrete logarithm problem is hard in the group used in the ECDHE, the zero-knowledge proof is secure and the compression function* f *of SHA-256 is an random oracle.*

We give the security proof in the Appendix A. It covers both a static semi-honest adversary, and also a static malicious adversary with abort.

6.1 Implementation

In DECO [27], the authors give a real example of using stock price API with DECO. By assessing the stock price API, a JSON output for a stock will be returned. In this paper, we consider a normal HTML webpage which provides some authenticated user information (the information is usually validated offline by a trustworthy webpage owner). In [27], it is mentioned that a university has the name and the date of birth for a student, but the university does not provides an open API for accessing this information. However, such an open API may not be always available. In this paper, we consider using the login account information of an utility company (e.g., electricity or gas) as the address proof, for applying some online financial services (e.g., challenger bank/virtual bank account, credit card or loans).

As an example, we first demonstrate that our DIDO implementation can be used to open a popular TLS 1.3 website, and then show that DIDO can serve as the address proof for user in some utility websites.

Some popular TLS 1.3 websites. We test our DIDO on some popular websites which support TLS 1.3. Our DIDO can successfully access the websites and the verifier is able to decrypt the payloads accordingly. We show the results in Table 4. It can be seen that there are some web pages with longer running time, and may have a big differences between LAN and WAN. The running time discrepancy is affected by the length of the content of the web page, and the response time of the server.

Address Proof. In a webpage of a utility company (e.g., electricity or gas), a user can use his username and password to login to his account and display his current address. It is common for a web server to send a cookie file after entering the username and password. By using the cookie file, the client browser can open the web page containing personal information. We test our implementation using a real gas company's web page.

1. The prover obtains the cookie file by the username and password (without using DIDO). For example, he can use the cURL command shown in Fig. 12 to obtain such cookie.txt.

Table 4: Running time of our DIDO implementation on different websites.

	$\mathcal{P}$ and $\mathcal{V}$ in LAN	$\mathcal{P}$ and $\mathcal{V}$ in WAN
www.google.com	4.266s	7.848s
www.youtube.com	6.384s	27.233s
github.com	4.717s	12.594s
www.microsoft.com	4.984s	6.275s
yahoo.com	4.468s	6.650s
wikipedia.org	4.338s	6.447s
easychair.org	5.665s	6.928s

```
rustls > src > $ curl.sh
1   curl --cookie-jar ./rustls/src/cookie.txt 'https://eservice.    gas.com/EAccount/Login/SignIn' \
2   --data-raw 'LoginID=          &password=           ' \
3   --compressed
```

Fig. 12: Command for getting the cookie file using LoginID and password.

2. The prover runs the post command in Fig. 13 to https://eservice.■■■■gas.com with our DIDO implementation, by putting the cookie file in the red box.

```
POST /NewsNotices/GetNewsNoticeAsyncNew HTTP/1.0\r\nHost: {}\r\nCookie: {}\r\nContent-Length:
20\r\nContent-Type: application/x-www-form-urlencoded\r\n\r\naccountNo=
```

Fig. 13: POST command for running rustls with our implementation of DIDO. The cookie file should be put in the red box.

3. An HTML file is returned by the server via TLS 1.3. It is encrypted by the key derived from SATS in Fig. 3. As discussed in section 5, SATS is only known to the verifier. Hence, the prover simply forwards the encrypted HTML to the verifier. The verifier uses the derived key to decrypt the HTML file (e.g., using AES-GCM). The user address is shown in the HTML file using the JSON format, as shown in Fig. 14.

In order to further test the flexibility of our approach, we also try it from an electricity company (https://services.■■■.com.■■) providing service to the same residential address. Finally, we obtain the HTML page in Fig. 15.

In this HTML page, the name of the account owner is also included. Hence, the account owner can use this HTML page as his address proof by using DIDO.

Further Implementation Details. In the paragraph of section 4.2, we propose to pick k_v until the v-coordinate of $x_v G$ is positive. This solution involves more engineering work. For the ease of testing, our sample code simply runs the ECtF+ protocol. With 50% probability, it fails and re-runs it again until it succeeds. The expected number of times of running the ECtF+ protocol is:

$$E[\#\mathsf{ECtF+}] = 0.5 * 1 + 0.25 * 2 + 0.125 * 3 + \ldots = 2.$$

Performance. We use the same setting in Section 4.1 for testing the performance of our DIDO implementation in both the LAN and the WAN settings. The running time is shown in Table 5. We can see that the verifier $\mathcal{V}$ can receive the HTML page from the server within 10s.

7 Conclusion

In this paper, we propose a new design DIDO for decentralized oracle with TLS 1.3. We provide support to X25519, propose a 2PC key scheduling and optimize the circuits for 2PC.

```
output.html
1  HTTP/1.1 200 OK
2  Cache-Control: no-store,no-cache
3  Pragma: no-cache
4  Content-Type: application/json; charset=utf-8
5  Server: Kestrel
6  X-Frame-Options: SAMEORIGIN
7  Strict-Transport-Security: max-age=31536000
8  X-Content-Type-Options: nosniff
9  X-Powered-By: ASP.NET
10 X-S: B
11 Server-Timing: dtSInfo;desc="0", dtRpid;desc="1633937757"
12 Set-Cookie: dtCookie=v_4_srv_1_sn_13348EE4CD358EBD01C3395D4A255E07_perc_100000_ol_0_mul_1_app-3Ad2243ac7e97f2a62_1; Path=/; Domain=.gas.com
13 Date: Fri, 18 Mar 2022 14:03:56 GMT
14 Connection: close
15 Set-Cookie: incap_ses_798_1579342=VwyWEt6uU2WsDCQRVhETC0yRNGIAAAAAq3OYsz53fAZJzGjGa0KNEA==; path=/; Domain=.gas.com
16 X-CDN: Imperva
17 X-Iinfo: 4-33923495-33923788 NNNN CT(5 7 0) RT(1647612227206 8792) q(0 1 1 0) r(2 2) U5
18
19 {"accountno":"","accountNoStatus":"A","gassupplyaddress":"FLAT 1, 10/F","bil
   lbalance":"-","billdate":null,"strbilldate":null,"lastpayment":null,"lastpaymentdate":null,"gasBillCodeContent":"","strlastpaymentdate":null,"qrCodeTex
   t":null,"isQrCodeExist":false,"currentAccountBalance":"0.00","updatedDate":"2022-03-17T22:03:56.7685095+08:00","strUpdatedDate":"2022-03-17","isIbillSe
   rvice":true,"isAutoPay":null,"isOverdueBill":null,"isNewAccount":true,"isHomeUsers":true,"billDueDate":null,"is150Indicator":null,"payAmountOverAutopay
   limit":null,"billAmountDue":"-","autoPayAmt":"-","payAmt":"-","isOpenAutomaticTransfer":true}
```


Fig. 14: The HTML file sent from the gas company and decrypted by the verifier.


```
output.html
1  HTTP/1.1 200 OK
2  Date: Thu, 31 Mar 2022 11:07:26 GMT
3  Content-Type: application/json; charset=utf-8
4  Transfer-Encoding: chunked
5  Connection: keep-alive
6  Cache-Control: no-cache, no-store, must-revalidate, no-store, must-revalidate
7  Pragma: no-cache
8  Expires: -1
9  Vary: Accept-Encoding
10 X-Frame-Options: SAMEORIGIN
11 X-Content-Type-Options: nosniff
12 Strict-Transport-Security: max-age=31536000
13 CF-Cache-Status: DYNAMIC
14 Expect-CT: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"
15 Set-Cookie: __cfruid=39ba4aced3233d50c49e627f830369930ccc6933-1648724846; path=/; domain=.; HttpOnly; Secure; SameSite=None
16 Server: cloudflare
17 CF-RAY: 6f489f0a98bf3cbe-HKG
18
19 46a
20 {"caNo":"","addr":"FLAT 01,10/F,","displayName":"","
   ValidFrom":null,"ValidTo":null,"QR_Code":null,"NextDueDate":null,"LastBillNumber":null,"LastBillAmount":null,"DunningAmount":"0.00","TotalDeposit":null,"D
   epositDueDate":null,"FirstBillDate":null,"OverDue_Indicator":"false","PayButton_Indicator":"fa
   lse","PaymentMethod":null,"isAutoPay":false,"lastBillPdf":"/Service/ServiceGetBillingPdfFile.ashx?caNo=\u0026Filename=11255639026.pdf\u0026bfIn
   dicator=\u0026tranDate=20220226000000","printDoc":"11255639026","payBtnColor":"","payBtnUrl":null,"statusLabel":"","accType":"rt","isMulti":"","MessageTyp
   e":"1","alertMsgData":[{"alertMessageType":"ReceivedPayment","payment":"Direct Debit","paymentAmount":"706.00","payDate":"20220322000000","billReadyDate":
   null,"startDate":null,"endDate":null,"AddInfoMsg":null,"ppsResult":null}],"ErrorCode":"","ErrorMsg":"","Tracer":["CallAPI_CA_BillSummary :
   356.6903","CallAPI_CA_BillPaymentHistory : 214.4914","CallAPI_AppProgressTracking_ProgressCheck : 948.2947","E2E : 1521.3333"]}
21 0
```


Fig. 15: The HTML file sent from the electricity company and decrypted by the verifier.

Table 5: Running time of our DIDO implementation on different utility companies.

	$\mathcal{P}$ and $\mathcal{V}$ in LAN	$\mathcal{P}$ and $\mathcal{V}$ in WAN
Gas company	4.003s	6.770s
Electricity company	6.303s	8.456s

References

1. Tlsnotary - a mechanism for independently audited https sessions. White paper (September 2014), `https://tlsnotary.org/TLSNotary.pdf`
2. Al Fardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the tls and dtls record protocols. In: 2013 IEEE Symposium on Security and Privacy. pp. 526–540 (2013). `https://doi.org/10.1109/SP.2013.42`
3. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von neumann architecture. In: Fu, K., Jung, J. (eds.) USENIX 2014. pp. 781–796. USENIX Association (2014)
4. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001. pp. 136–145. IEEE Computer Society (2001)
5. Castagnos, G., Laguillaumie, F.: Linearly homomorphic encryption from DDH. In: Nyberg, K. (ed.) CT-RSA 2015. Lecture Notes in Computer Science, vol. 9048, pp. 487–505. Springer (2015)
6. Cavage, M., Sporny, M.: Signing http messages. `https://tools.ietf.org/id/draft-cavage-http-signatures-12.html` (2019)
7. Dierks, T., Rescorla, E.: The transport layer security (tls) protocol version 1.2. RFC 5246, RFC Editor (August 2008), `https://datatracker.ietf.org/doc/html/rfc5246`
8. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol. J. Cryptol. **34**(4), 37 (2021)
9. Duong, T., Rizzo, J.: Here come the ⊕ ninjas. Unpublished manuscript (2011), `https://tlseminar.github.io/docs/beast.pdf`
10. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) CCS 2018. pp. 1179–1194. ACM (2018)
11. Josefsson, S., Liusvaara, I.: Edwards-curve digital signature algorithm (eddsa). RFC 8032, RFC Editor (January 2017), `https://datatracker.ietf.org/doc/html/rfc8032`
12. Keller, M.: Mp-spdz: A versatile framework for multi-party computation. In: CCS 2020. p. 1575–1590. Association for Computing Machinery, New York, NY, USA (2020). `https://doi.org/10.1145/3372297.3417872`
13. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. Lecture Notes in Computer Science, vol. 5126, pp. 486–498. Springer (2008)
14. Langley, A., Hamburg, M., Turner, S.: Elliptic curves for security. RFC 7748, RFC Editor (January 2016), `https://datatracker.ietf.org/doc/html/rfc7748`
15. Nilsson, A., Bideh, P.N., Brorsson, J.: A survey of published attacks on intel SGX. CoRR **abs/2006.13598** (2020), `https://arxiv.org/abs/2006.13598`
16. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT '99. Lecture Notes in Computer Science, vol. 1592, pp. 223–238. Springer (1999)
17. Rescorla, E.: The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor (August 2018), `https://datatracker.ietf.org/doc/html/rfc8446`
18. Ritzdorf, H., Wüst, K., Gervais, A., Felley, G., Capkun, S.: TLS-N: non-repudiation over TLS enablign ubiquitous content signing. In: NDSS 2018. The Internet Society (2018)
19. Smith, B.: Ring - Safe, fast, small crypto using Rust. `https://github.com/briansmith/ring` (2022)
20. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit` (2016)
21. Wang, X., Malozemoff, A.J., Katz, J.: Faster secure two-party computation in the single-execution setting. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017. pp. 399–424. Springer International Publishing, Cham (2017)
22. Wang, X., Ranellucci, S., Katz, J.: Authenticated garbling and efficient maliciously secure two-party computation. In: CCS 2017. p. 21–37. Association for Computing Machinery, New York, NY, USA (2017). `https://doi.org/10.1145/3133956.3134053`
23. Xue, H., Au, M.H., Xie, X., Yuen, T.H., Cui, H.: Efficient online-friendly two-party ECDSA signature. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS '21. pp. 558–573. ACM (2021)
24. Yao, A.C.: Protocols for secure computations (extended abstract). In: FOCS '82. pp. 160–164. IEEE Computer Society (1982)

25. Yasskin, J.: Signed http exchanges. Internet-Draft: draft-yasskin-http-origin-signed-responses-latest (2022)
26. Zhang, F., Cecchetti, E., Croman, K., Juels, A., Shi, E.: Town crier: An authenticated data feed for smart contracts. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) CCS 2016. pp. 270–282. ACM (2016)
27. Zhang, F., Maram, D., Malvai, H., Goldfeder, S., Juels, A.: DECO: liberating web data using decentralized oracles for TLS. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS '20. pp. 1919–1938. ACM (2020)

A Security of DIDO

Proof. In the proof, we will show that for any real-world adversary $\mathcal{A}$, an ideal simulator Sim can be constructed. The ideal execution with is indistinguishable from the real execution with $\mathcal{A}$ for all environments $\mathcal{Z}$.

As aforementioned, we assumed $\mathcal{S}$ is honest throughout the protocol. Therefore, we only consider $\mathcal{A}$ corrupts either $\mathcal{P}$ or $\mathcal{V}$ in the proof. Since $\mathcal{V}$ does not have any private input, a semi-honest $\mathcal{P}$ cannot gain any private information of $\mathcal{V}$. We have prover-integrity since $\mathcal{P}$ is semi-honest and follows the protocol. Thus we only need the ideal-world simulators for corrupted $\mathcal{V}$.

In addition, we give the modification of the proof for malicious security in a red box. We note that we can achieve malicious security if the underlying 2PC-HMAC protocol is secure in the malicious setting.

Corrupted $\mathcal{V}$. As the verifier is corrupted, we are interested in showing the privacy guarantees. The simulator Sim proceeds as follows:

1. Sim internally runs $\mathcal{A}$ and $\mathcal{F}_{\mathsf{2PC}}$ to simulate the real-world interaction with the prover $\mathcal{P}$. Sim forwards any input z from $\mathcal{Z}$ to $\mathcal{A}$.
2. Sim forwards the Query to $\mathcal{F}$ from $\mathcal{A}$, and instructs $\mathcal{F}$ send them to $\mathcal{P}$.
3. $\mathcal{P}$ sends θ_s to $\mathcal{F}$, then $\mathcal{F}$ responds with (sid, Q, R) to $\mathcal{P}$. Sim receives $(\mathsf{sid}, \mathcal{S})$ from $\mathcal{F}$ and learns the record sizes of Q and R, denote as $|Q|$ and $|R|$.
4. Sim runs 3P-DH as $\mathcal{P}$ upon request from $\mathcal{A}$. Sim and $\mathcal{A}$ runs the Step 1-2 of 3P-DH. Sim uses the extractor of the zero-knowledge proof to get x_v from π_v. After Step 2, $x_pG + x_vG$ is used as the client *key_share* of TLS 1.3 ClientHello message. Sim generates the rest of ClientHello and sends $(\mathsf{sid}, \mathcal{S}, \mathsf{ClientHello})$ to $\mathcal{F}$.
5. $\mathcal{F}$ returns ServerHello from $\mathcal{S}$, which contains the (unencrypted) server *key_share* yG. Sim continues with the Step 3-4 of 3P-DH with $\mathcal{A}$, learning its additive share u_p at the end. Since Sim knows x_v and x_p, he can also compute the x-coordinate u of $(x_p + x_v)yG$ by himself. Then Sim computes $u_v = u - u_p$.
6. Sim starts the 2PC key scheduling as $\mathcal{P}$ with private input u_p. Sim sends his private input to $\mathcal{F}_{\mathsf{2PC}}$ when 2PC-HMAC is used, and obtains his output. After running the 2PC key scheduling, Sim obtains CHTS, SHTS and CATS. Sim derives the key $\mathsf{tk}_{\mathsf{shs}}$ from SHTS for decrypting the rest of ServerHello, and completes the rest of the handshake (e.g., *ChangeCipherSpec, Finished* in TLS 1.3).
7. Since Sim knows both u_v and u_p, he can also compute the key scheduling by himself and obtains CATS' and SATS'. Sim picks a random $Q' \leftarrow \{0,1\}^{|Q|}$ and derives the key $\mathsf{tk}_{\mathsf{capp}}$ from CATS. Sim encrypts Q' using tk_{capp} to obtain the ciphertext $\hat{Q}$. If CATS' $\neq$ CATS, Sim sets $\hat{R}$ as the TLS 1.3 decryption error message. Otherwise, Sim picks a random response R' with length $|R|$, and then encrypts R' with the key tk_{sapp} from SATS' to obtain $\hat{R}$. Sim sends $(\hat{Q}, \hat{R})$ to $\mathcal{A}$ and outputs whatever $\mathcal{A}$ outputs.

Now we discuss that the ideal execution with Sim is indistinguishable from the real one with $\mathcal{A}$.

- **Hybrid** H_1 is the real-world execution of $\mathsf{P}_{\mathsf{DIDO}}$.

- **Hybrid** H_2 is the same as H_1, except that Sim internally simulates $\mathcal{F}_{\mathsf{2PC}}$. By using the extractor of the zero-knowledge proof, Sim can extract x_v from π_v. Sim also invokes $\mathcal{F}$ and receives $(\mathsf{sid}, \mathsf{Stmt}(R), \mathcal{S})$. It also learns the sizes of Q and R. Due to the perfect simulation of ideal functionality, H_1 and H_2 are indistinguishable.
- **Hybrid** H_3 is the same as H_2, except that $\mathcal{P}$ is simulated by Sim. To simulate the key uses in the sequential 2PC-HMAC invocations, Sim samples x_p and uses x_pG to derive a share of the ECDHE secret u_p.
  By using x_v and x_p, Sim can compute the ECDHE secret u and also $u_v = u - u_p$. Sim can generate all keys in key scheduling and completes the handshake messages. Then, Sim picks a random $Q' \in \{0,1\}^{|Q|}$ and a random $R' \in \{0,1\}^{|R|}$. Afterwards, Sim sends the encryption of Q' and R' to $\mathcal{A}$.

The reasons why $\mathcal{A}$ cannot distinguish between the real and ideal executions:

- The number of invocations of 2PC-HMAC is equal since the input sizes are equal.
- $\mathcal{A}$ learns one SHA-2 hash of the input message which is like a random oracle in each invocation of 2PC-HMAC and HMAC.
- If the input provided by $\mathcal{V}$ in 2PC-HMAC is correct, all messages should verify and the protocol should proceed to the next step in both the real and ideal world. The checking time should be indistinguishable from the real world.

- $\mathcal{A}$ can act maliciously in any of the following steps: (1) Deviate from the Step 4 of 3P-DH. If so, $\mathcal{P}$ cannot obtain a valid share of DHE. It will not pass the handshake checking with $\mathcal{S}$. (2) Deviate from the 2PC-HMAC for generating HS, CHTS or SHTS. If so, $\mathcal{P}$ cannot obtain valid handshake keys. It will not pass the handshake checking with $\mathcal{S}$. (3) Deviate from the 2PC-HMAC for generating dHS, MS or CATS. If so, $\mathcal{P}$ cannot obtain a valid client application data key. $\mathcal{S}$ cannot decrypt the query and will return the TLS 1.3 error message. (4) Deviate from the 2PC-HMAC for generating SATS. $\mathcal{A}$ cannot obtain any information of $\mathcal{P}$ in this case. For all cases, if the 2PC-HMAC is secure in the malicious setting, $\mathcal{A}$ cannot obtain the input of $\mathcal{P}$ even if he deviates from the protocol.

- The encryption sizes are equal and indistinguishable since $|Q'| = |Q|$ and $|R'| = |R|$.
- At the end, $\mathcal{A}$ receives the same output as in real execution.

Corrupted $\mathcal{P}$. We show the prover-integrity guarantee in the malicious model. If $\mathcal{V}$ receives $(\mathsf{Stmt}(R), \mathcal{S})$, $\mathcal{P}$ must have input some θ_s such that $\mathcal{S}(\mathsf{Query}(\theta_s)) = \mathsf{Stmt}(R) = R$.
The Sim works as follows given a real-world PPT adversary $\mathcal{A}$.

1. Sim runs $\mathcal{A}$ and $\mathcal{F}_{\mathsf{2PC}}$ internally. Sim forwards any input z from $\mathcal{Z}$ to $\mathcal{A}$ and records all the traffic going to and from $\mathcal{A}$. Sim records and forwards its private input θ_s to $\mathcal{A}$.
2. Sim runs the 3P-DH as $\mathcal{V}$ upon the request of $\mathcal{A}$. During the process, Sim forwards the output message m from $\mathcal{A}$, which is intended for $\mathcal{S}$, to $\mathcal{F}$ as $(\mathsf{sid}, \mathcal{S}, m)$ and forwards (sid, m') to $\mathcal{A}$ if it receives any messages m' from $\mathcal{F}$. Sim learns x_pG, x_v and u_v. Sim uses the extractor of the zero-knowledge proof π_p to obtain x_p.
3. Sim runs 2PC key scheduling as $\mathcal{V}$ upon request from $\mathcal{A}$, using u_v as input. Similarly, Sim uses $\mathcal{F}_{\mathsf{2PC}}$ as a sub-routine to run all 2PC-HMAC. Sim learns the secret SATS.
4. Sim records all the messages between $\mathcal{A}$ and $\mathcal{S}$ in the Query phase. Note that these include ciphertext encrypted with keys derived from CHTS, SHTS and CATS.
5. By using x_p and x_v, Sim can run the key scheduling all by himself and derives CHTS', SHTS', CATS' and SATS'.
6. $\mathcal{A}$ sends $(\mathsf{sid}, m_{\mathsf{sapp}})$. Sim uses SATS to derive a key and to decrypt m_{sapp}. Sim aborts if the decryption fails.
7. Sim checks the followings: (1) In Step 2 of 3P-DH, $\mathcal{A}$ sends $(x_p+x_v)G$ to $\mathcal{S}$ as the client key share. (2) The client/server finish message can be decrypted by the keys derived from CHTS'/SHTS'. (3) The encrypted request $\hat{Q}$ can be decrypted to θ_s by the key

derived from CATS'. (4) SATS $\neq$ SATS'. If all of the above checks passed, Sim sends θ_s to $\mathcal{F}$ and instructs $\mathcal{F}$ to send the output to $\mathcal{V}$. Sim outputs whatever $\mathcal{A}$ outputs.

Now we discuss that the ideal execution with Sim is indistinguishable from the real one with $\mathcal{A}$.

- **Hybrid** H_1 is the real-world execution of $\mathsf{P}_{\mathsf{DIDO}}$.
- **Hybrid** H_2 is the same as H_1, except for step 1 that Sim internally simulates $\mathcal{A}$ and $\mathcal{F}_{\mathsf{2PC}}$. Sim records the private θ_s input and forwards it to $\mathcal{A}$. For each step of $\mathsf{P}_{\mathsf{DIDO}}$, Sim forwards all messages between $\mathcal{A}$ and $\mathcal{V}$, and $\mathcal{A}$ and $\mathcal{S}$, as in the real execution. Due to the perfect simulation of ideal functionality, H_1 and H_2 are indistinguishable.
- **Hybrid** H_3 is the same as H_2, except that Sim internally simulates $\mathcal{V}$. To simulate the keys used, Sim samples $\hat{x}_v$ and uses $\hat{x}_v G$ to derive a share u_v, which is used in the sequential 2PC-HMAC invocations. Sim obtains SATS after the 2PC Key Scheduling with input u_v. Upon receiving $(\mathsf{sid}, m_{\mathsf{sapp}})$ from $\mathcal{A}$, Sim uses the key derived from SATS to decrypt m_{sapp} and obtains the response. Sim aborts if the decryption fails. The indistinguishability between H_2 and H_3 is trivial because $\hat{x}_v$ is uniformly random.
- **Hybrid** H_4 is the same as H_3, except that Sim extracts x_p from the zero-knowledge proof π_p and runs the key scheduling by himself. After that, Sim runs the checking in the above Step 7 and it aborts if the checking fails.

There are two reasons that Sim may abort: (1) m_{sapp} from $\mathcal{A}$ is not originally from $\mathcal{S}$. (2) the keys used by $\mathcal{S}$ is not the same as the keys derived by 2PC-HMAC. We now show that it would trigger $\mathcal{V}$ to abort as well with overwhelming probability.

- Assuming that DL is hard in the ECDHE group used. $\mathcal{A}$ cannot learn $\hat{x}_v$ from $\hat{x}_v G$. By the zero-knowledge property, $\mathcal{A}$ cannot learn $\hat{x}_v$ from π_v. If $\mathcal{A}$ malicious selects the client key share (denoted $\hat{Y}$) correlated with $\hat{x}_v G$, it would have to find the discrete logarithm of $\hat{Y} - \hat{x}_v G$ to run the 3P-DH. Without such value, the output shares u_v and u_p of 3P-DH would fail to derive keys to complete the handshake with an honest server, except with negligible probability.
- By the security of 2PC-HMAC, $\mathcal{A}$ cannot learn SATS. By the security of AES-GCM, without the knowledge of the key SATS, $\mathcal{A}$ cannot create a valid m_{sapp} that can be decrypted successfully without abort.
- If $\mathcal{A}$ ignores the input $\hat{x}_v G$ from Sim in 3P-DH and generates it by his own, he has a valid server response encrypted with keys derived from some SATS*. By the security of 2PC-HMAC, $\mathcal{A}$ cannot force Sim to output SATS = SATS* which is chosen by $\mathcal{A}$.

It remains to show that H_4 is exactly the same as the ideal execution. Due to Step 7, $\mathcal{F}$ delivers $(\mathsf{sid}, \mathsf{Stmt}(R), \mathcal{S})$ to $\mathcal{V}$ only if $\exists \theta_s$ such that R is the response from $\mathcal{S}$ to $\mathsf{Query}(\theta_s)$.

$\square$

B Discussion among the limitation from existing 2PC protocols

As aforementioned in Section 3.2 and 3.3, the main performance bottleneck of DIDO is the large number of 2PC-SHA256 in key computations such as CHTS and SHTS. Moreover, large amount of invocations of the existing 2PC-HMAC protocols will incur a running time larger than the TLS 1.3 timeout. The default TLS timeout is default in 10s according to the documentation of IBM[7] and 15s for Microsoft[8].

In DECO [27], it adopts the malicious model using emp-ag2pc [20, 22][9]. According to Table 6 of [22], one 2PC-SHA256 requires 2376ms. In our TLS 1.3 implementation, there are roughly 30 invocations of 2PC-SHA256 which require 71s. This running time is not only unacceptable

[7] https://www.ibm.com/docs/en/zos/2.5.0?topic=considerations-handshake-timer

[8] https://techcommunity.microsoft.com/t5/ask-the-directory-services-team/tls-handshake-errors-and-connection-timeouts-maybe-it-8217-s-the/ba-p/400501

[9] https://github.com/emp-toolkit/emp-ag2pc

for normal user, but also will trigger timeout in most server. A further ZK-SNARK for selective opening will make it even worse. The imbalance between the tight performance requirement to avoid timeout and the performance of existing 2PC protocols leads to the limitation of our DIDO protocol to become practical if we keep the malicious security setting. Therefore, we further compromise and make our protocol under the semi-honest security model, in which case we can drop the time-consuming NIZKs.

We emphasize that the architecture of DIDO (e.g., design of 2PC-key derivation, modification for Curve25519) supports the malicious security, but limited by the aforementioned performance bottleneck. In order to explain this, we will first discuss the performance of the existing 2PC protocols, and the possibility of the enhancement from semi-honest security model to a malicious one.

Performance of existing 2PC protocols. We give a comparison between existing 2PC protocols as follows to explain why we have such decision.

According to Table 1 of [12], in the existing implementations of 2PC-Garbled-Circuit, MP-SPDZ [12] and EMP-toolkit [21, 22] are the existing implementations with the best performance. We first consider using the newer MP-SPDZ. However, it does not support any online/offline operations. Hence, we choose the EMP-toolkit library which contains components with different security levels, such as emp-ag2pc for malicious security and emp-sh2pc for semi-honest security.

We adopt emp-sh2pc in our design since emp-ag2pc for malicious security will trigger a TLS1.3 timeout; in contrast, emp-sh2pc for semi-honest security will not. Accordingly, we further relax other parts of DIDO to the semi-honest model for optimization, including the removal of zero-knowledge proofs for MtA.

Possible improvement towards a malicious setting. To further enhance our security setting of DIDO to malicious model, adoption of both/either a 2PC protocol with outstanding performance and/or faster hardware may work. As aforementioned, the existing 2PC protocols which achieve malicious security setting cannot fulfil the performance requirement. Moreover, it is not likely that a faster hardware could have a 10x improvement in running time to avoid timeout. It is more likely to work if there is a faster 2PC-SHA256 by some algorithmical improvements. Therefore, the problem of implementing fast enough 2PC-SHA256 is still an open problem.