Lightweight Authentication of Web Data via Garble-Then-Prove

Xiang Xie [*] Kang Yang [†] Xiao Wang [‡] Yu Yu [§]

Abstract

Transport Layer Security (TLS) establishes an authenticated and confidential channel to deliver data for almost all Internet applications. A recent work (Zhang et al., CCS'20) proposed a protocol to prove the TLS payload to a third party, without any modification of TLS servers, while ensuring the privacy and originality of the data in the presence of malicious adversaries. However, it required maliciously secure Two-Party Computation (2PC) for generic circuits, leading to significant computational and communication overhead.

This paper proposes the garble-then-prove technique to achieve the same security requirement without using any heavy mechanism like generic malicious 2PC. Our end-to-end implementation shows 14× improvement in communication and an order of magnitude improvement in computation over the state-of-the-art protocol. We also show worldwide performance when using our protocol to authenticate payload data from Coinbase and Twitter APIs. Finally, we propose an efficient gadget to privately convert the above authenticated TLS payload to additively homomorphic commitments so that the properties of the payload can be proven efficiently using zkSNARKs.

1 Introduction

Transport Layer Security (TLS) [26, 55] is the most widely deployed cryptographic protocol for secure communication on the Internet. It provides end-to-end security against active attackers between a client, namely $\mathcal{C}$ and a TLS server, namely $\mathcal{S}$. However, if the client wants to use the TLS payload data in a different application, TLS does not guarantee the originality of the data. In particular, a malicious client could come up with a valid TLS transcript for any payload of its choice. The issue stems from the fact that the TLS protocol assumes that both client $\mathcal{C}$ and server $\mathcal{S}$ are honest, but in this new setting, the client can be malicious. For most websites, this is solved by having a user authenticate one website in connection with the other website that needs the data. Doing so under the client's authorization allows the two websites to share data directly and thus ensures no malicious client can break integrity. However, such a solution is not perfect. First, users are often forced to share more information than needed, e.g., to prove that their credit score is higher than a threshold, they need to share the score entirely. Second, this solution requires adding new web infrastructures, which could hinder the deployment, especially when connecting Web2 data to Web3 applications.

A recent work, DECO [71], proposed a solution that does not require any change on the TLS server side. From a high-level view, they ask a prover $\mathcal{P}$ (i.e., a user that intends to prove the originality of the data) and a verifier $\mathcal{V}$ (i.e., a third party) to jointly emulate the computation of the TLS client $\mathcal{C}$ who interacts with $\mathcal{S}$. Since neither $\mathcal{P}$ nor $\mathcal{V}$ ever holds TLS session keys, their capability is the same as man-in-the-middle attackers and thus cannot forge a valid TLS transcript for unauthorized data. In DECO, most of $\mathcal{C}$'s computation is emulated using a maliciously secure Two-Party Computation (2PC) protocol, which ensures that no derivation from the protocol can help the malicious party break the privacy or integrity requirement when interacting with $\mathcal{S}$. To prove statements on the TLS payload, $\mathcal{P}$ proves to $\mathcal{V}$ the correct decryption of the ciphertext (to obtain a plaintext) and desired statements on the plaintext using zkSNARKs [8, 10, 30, 33].

Generic 2PC protocols in the malicious setting have been studied extensively in the past decade (e.g., [15, 50, 51, 53, 59]). DECO used an implementation of the authenticated garbling [36, 43, 62, 67], the state-of-the-art malicious 2PC framework that significantly reduces the overhead compared to the semi-honest counterparts. However, even based on the latest advances [23, 27], the computation and communication cost of maliciously secure 2PC is still much higher than its semi-honest counterparts. Moreover, these protocols with malicious security often require storing preprocessed authen-

*PADO Labs, xiexiangiscas@gmail.com
†State Key Laboratory of Cryptology, yangk@sklc.org
‡Northwestern University, wangxiao@northwestern.edu
§Shanghai Jiao Tong University & Shanghai Qi Zhi Institute, yuyu@cs.sjtu.edu.cn

ticated triples, thus incurring a huge memory overhead. The complexity of the maliciously secure protocol also makes it difficult to implement and deploy such a protocol. As a result, the DECO protocol still requires 475 MB of communication to authenticate a 2KB-sized payload via TLS and more than 50 seconds to finish under a WAN network.

1.1 Our Contribution

In this paper, we design a new protocol for web-data authentication to third parties with improved efficiency. We propose the garble-then-prove technique that can realize a special class of two-party computation functionalities against malicious adversaries, with almost no overhead compared to their semi-honest counterparts. We elaborate on our key concepts and contributions below and refer to Section 3 for an overview of our core techniques.

Eliminating malicious 2PC via garble-then-prove. We avoid the use of maliciously secure 2PC, as a result of deeply understanding the features of authenticating web data in TLS. We observe that since $\mathcal{V}$ is the verifier, the security requirements for $\mathcal{V}$ and the prover $\mathcal{P}$ differ in many ways. **During** the secure TLS emulation, a corrupted $\mathcal{V}$ shall not learn the session keys as it immediately reveals $\mathcal{P}$'s private input; however, we can tolerate a corrupted $\mathcal{P}$ learning some information about the session keys: since $\mathcal{V}$ does not have long-term secrets, the damage is remediable. We only require $\mathcal{P}$'s cheating behavior to be identifiable by $\mathcal{V}$ later. **After** the completion of the joint TLS emulation, all of $\mathcal{V}$'s shares of the TLS secrets can be opened to $\mathcal{P}$ since $\mathcal{P}$ can no longer alter the TLS protocol. Simply put, our security requirement is as below: $\mathcal{P}$ and $\mathcal{V}$ start with inputs $x_{\mathcal{P}}$ and $x_{\mathcal{V}}$ respectively and shall get outputs $y_{\mathcal{P}}, y_{\mathcal{V}}$ such that $(y_{\mathcal{P}}, y_{\mathcal{V}}) = f(x_{\mathcal{P}}, x_{\mathcal{V}})$ for some two-output function f. If $\mathcal{P}$ cheats, it can replace the function to one of its own choice but $\mathcal{V}$ cannot cheat in any way. During the checking phase, $\mathcal{P}$ will be given $x_{\mathcal{V}}$ and $\mathcal{V}$ should be notified if $\mathcal{P}$ cheated during the evaluation phase.

To accomplish this task, $\mathcal{P}$ first sends $\mathcal{V}$ a garbled circuit for f; they also use an OT with malicious security to let $\mathcal{V}$ get garbled labels on its input. Two parties then can obtain their outputs but there is no way to ensure correctness. For that, we ask $\mathcal{P}$ to commit to $\mathcal{V}$ its input $x_{\mathcal{P}}$ and output $y_{\mathcal{P}}$. Now, $\mathcal{V}$ has shares $x_{\mathcal{V}}, y_{\mathcal{V}}$ and commitments of $x_{\mathcal{P}}, y_{\mathcal{P}}$. After $\mathcal{P}$ gets $x_{\mathcal{V}}$, thus also $y_{\mathcal{V}}$, $\mathcal{P}$ can use a Zero-Knowledge (ZK) protocol to prove that $(y_{\mathcal{P}}, y_{\mathcal{V}}) = f(x_{\mathcal{P}}, x_{\mathcal{V}})$ w.r.t. the committed values. $\mathcal{P}$ could launch a selective failure attack on $x_{\mathcal{V}}$ (leaking one-bit of information), but it is meaningless since $x_{\mathcal{V}}$ is always given to $\mathcal{P}$ in the proving phase. For obvious reasons, we refer to this technique as *garble-then-prove*. This technique can be also applied in, e.g., QUIC [24, 39], OAuth [35] and OpenID Connect [58], to authenticate web data.

TLS-specific protocol optimization. Building on the above idea, we further optimize other TLS building blocks in various ways. For example, we show how to carefully select values to reveal, without providing any party an extra capacity, during the derivation of TLS session keys, leading to a more than 2-fold reduction in handshake circuit size. We also pull the computation of the Galois Message Authentication Code (GMAC) tags out of circuits and instead use Oblivious Linear Evaluation with errors (OLEe) [1] to compute additive sharings of the powers of a random element needed for GMAC, reducing the cost of GMAC computation by more than two orders of magnitude. Doing so would allow the adversary to gain one bit of information of the TLS session key, but that would not reduce the overall concrete security, for a reason similar to prior works, e.g., [44, 67].

Efficient commitment conversion. To prove statements on the TLS data using zkSNARKs, DECO embeds the TLS ciphertext into the statements and then proves in ZK the correctness of decryption. For our protocol, we use the recent vector OLE (VOLE) based interactive zero-knowledge proofs [7, 28, 63, 66] during the garble-then-prove execution. This means that at the end of the protocol, two parties hold information-theoretic MACs (IT-MACs) on each bit of the query and response involved in the TLS. One could prove statements using VOLE-based ZK proofs or, alternatively, convert them to commitments friendly to zkSNARKs. First, we convert IT-MACs over $\mathbb{F}_2$ to IT-MACs over $\mathbb{F}_q$, ensuring the values are consistent. This protocol can be viewed as a special version of zero-knowledge via garbled-circuit protocol [41] over garbling of Boolean-to-arithmetic identity gates [6]. This makes the cost conversion in the malicious case almost the same as the semi-honest setting. Then we convert arithmetic IT-MACs to zkSNARK-friendly commitments, which can be achieved with high efficiency, since both representations are additive-homomorphic. In this way, without using zkSNARKs, we can convert the plaintext query and response to additively homomorphic commitments, which can then be connected to various zkSNARKs, e.g., [17, 18, 20].

Full-fledged implementation. We implemented our protocol and report detailed performance in Section 5. Our protocol outperforms DECO by more than an order of magnitude: 14× improvement in communication and 7.5× to 15× improvements in running time. We also push through the last mile to connect our implementation with real-world APIs connected via TLS. In Section 5.3, we include two examples of using our protocol to authenticate API results from Coinbase and Twitter. We report the performance when the prover is located in 18 cities worldwide with various network conditions. We also show a summary of the performance in Table 1, where we can see that the whole protocol only takes around 7 seconds (4 seconds of online time) when a user in Tokyo proving to a verifier in California about its Coinbase/Twitter API payload.

[1]OLEe provides a weaker security in which the malicious party can introduce an error into the OLE output, but it can be generated more efficiently.

Region of prover $\mathcal{P}$	Oregon	Virginia	Milan	Singapore	Tokyo
Coinbase	1.66 (2.43)	2.85 (4.98)	6.47 (11.9)	6.05 (11.7)	3.94 (7.35)
Twitter	0.94 (1.71)	2.08 (4.10)	5.21 (10.8)	5.78 (11.7)	3.56 (7.12)

Table 1: **Performance summary of our protocol.** All numbers are reported in seconds, based on the Coinbase API to query account balance (426-byte query and 5701-byte response) and the Twitter API to query the number of followers (587-byte query and 894-byte response). Both online time and total time (in parentheses) are reported. Verifier $\mathcal{V}$ is always located at California.

2 Preliminaries

We describe the TLS building blocks and model the security of authenticating web data. The cryptographic preliminaries to comprehend our protocol are described in Section A.

Notation. We use λ to denote the computational security parameter. We use $x \leftarrow S$ to denote that sampling x uniformly at random from a finite set S. For an algorithm A, we use $y \leftarrow \mathsf{A}(x)$ to denote the operation of running A on input x and setting y as the output. We will use bold lower-case letters like $\boldsymbol{x}$ for column vectors, and denote by x_i the i-th component of $\boldsymbol{x}$ with x_1 the first entry. For $a, b \in \mathbb{N}$, we write $[a,b] = \{a,\ldots,b\}$. We write $\mathbb{F}_{2^\lambda} \cong \mathbb{F}_2[X]/f(X)$ for some monic, irreducible polynomial $f(X)$ of degree λ. Depending on the context, we use $\{0,1\}^\lambda$, $(\mathbb{F}_2)^\lambda$ and $\mathbb{F}_{2^\lambda}$ interchangeably, and thus addition in $(\mathbb{F}_2)^\lambda$ and $\mathbb{F}_{2^\lambda}$ corresponds to XOR in $\{0,1\}^\lambda$ and a string $a \in \{0,1\}^\lambda$ is also a vector in $(\mathbb{F}_2)^\lambda$. For a bit-string x, we use $\mathsf{lsb}(x)$ to denote the least significant bit of x. For a prime p, we denote by $\mathbb{Z}_p$ a finite field.

We use $[x]_p = (x_{\mathcal{P}}, x_{\mathcal{V}})$ to denote an additive secret sharing of x over $\mathbb{Z}_p$ between $\mathcal{P}$ and $\mathcal{V}$ holding $x_{\mathcal{P}}$ and $x_{\mathcal{V}}$ respectively. When the field is $\mathbb{F}_{2^{128}}$, we denote by $[x]_{2^{128}}$. For details of additive secret sharings, we refer to the reader for Section A. Let $[\![x]\!] = (x, \mathsf{M}[x], \mathsf{K}[x])$ be an Information-Theoretic Message Authentication Code (IT-MAC) such that $\mathsf{M}[x] = \mathsf{K}[x] + x \cdot \Delta$, where the message x and MAC tag $\mathsf{M}[x]$ are held by a party $\mathcal{P}$, and keys $\mathsf{K}[x], \Delta$ are obtained by another party $\mathcal{V}$. We give more details of IT-MACs in the full version [65, Section A].

2.1 TLS Building Blocks

Transport Layer Security (TLS) is a family of protocols that guarantee privacy and integrity of data between a client $\mathcal{C}$ and a server $\mathcal{S}$. It consists of two protocols: (a) the handshake protocol in which handshake secrets are established and the secrets are in turn used to generate application keys; (b) the record protocol where data is transmitted with confidentiality and integrity via encrypting and authenticating the data with the application keys. Our protocol focuses on authenticating web data for TLS 1.2 [26], and is able to be extended to TLS 1.3 [55] that is shown in Section 4.3, where both of TLS 1.2 and TLS 1.3 adopt HMAC to derive secrets and keys. [2] While TLS provides different modes, we focus on the following most popular modes:

$$\mathsf{ECDHE\_RSA\_AES128\_GCM\_SHA256}$$
$$\mathsf{ECDHE\_ECDSA\_AES128\_GCM\_SHA256},$$

where the hash function H is instantiated by $\mathsf{SHA256}$, and a stateful Authenticated Encryption with Associated Data (AEAD) scheme is instantiated by $\mathsf{AES128}$ in the GCM mode. ECDHE adopts the elliptic-curve Diffie-Hellman (DH) key exchange protocol to establish ephemeral secrets.

Our protocol is easy to be extended to support that AEAD scheme is instantiated by $\mathsf{AES256\_GCM}$ and H is replaced with $\mathsf{SHA384}$, and also allows one to use other digital signature (e.g., DSA). Besides, our protocol can be straightforwardly extended to support ECDH in which the server uses a static DH value (rather than an ephemeral DH value). We did not optimize our protocol to realize the CBC mode in TLS 1.2, since this mode has been demonstrated to be vulnerable to the timing attack against several TLS implementations [5], and the GCM mode is preferred over CBC [2]. In addition, TLS 1.3 did not support the CBC mode any more. Our garble-then-prove approach can be also generalized to other modes such as $\mathsf{CHACHA20\_POLY1305\_SHA256}$ and $\mathsf{AES128\_CCM\_SHA256}$. In the full version [65], we describe the TLS 1.2 protocol in detail. Below, we describe several key building blocks used in the TLS protocol.

HMAC. Given a key k and a message m as input, the well-known pseudo-random function HMAC is defined as follows:

$$\mathsf{HMAC}(k,m) = \mathsf{H}(k \oplus \mathrm{opad}, \mathsf{H}(k \oplus \mathrm{ipad}, m)),$$

where opad and ipad are two public strings with length of 512 bits (i.e., the repeated bytes of 0x36 and 0x5C respectively). Here we always assume that k has at most 512 bits, which is the case for TLS. When the bit-length of k is less than 512, it will be padded with 0 to achieve 512 bits. As described above, we focus on considering that H is instantiated by $\mathsf{SHA256}$. In particular, $\mathsf{SHA256}$ adopts the Merkle-Damgård structure with block size of 512 bits, and uses f_{H} as the one-way compression function with output length of 256 bits. For example, $\mathsf{H}(m_1, m_2)$ is computed as $f_{\mathsf{H}}(f_{\mathsf{H}}(IV_0, m_1), m_2)$ where $m_1, m_2 \in \{0,1\}^{512}$ and IV_0 is a fixed initial vector.

Key derivation. Here we focus on the Pseudo-Random Function (PRF) in TLS 1.2 [26], where the PRF is used to derive

[2] For now, about 77%∼79% websites use TLS 1.2, while about 9%∼20% websites adopt TLS 1.3 [1].

handshake secrets and application keys and adopts HMAC as its core. TLS 1.3 [55] adopts the HKDF function [46,47] as its key derivation function, where this function is also based on HMAC. We refer the reader to Section 4.3 for the details of HKDF. Specifically, the PRF function with output length ℓ in TLS 1.2 is defined below:

$$\mathsf{PRF}_\ell(k, label, msg) = \mathsf{HMAC}(k, M_1 \| label \| msg) \| \ldots \| \mathsf{HMAC}(k, M_{n-1} \| label \| msg) \| \mathsf{Trunc}_m(\mathsf{HMAC}(k, M_n \| label \| msg)),$$

where $n = \lceil \ell/256 \rceil$, $m = \ell - 256 \cdot (n-1)$, $M_1 = \mathsf{HMAC}(k, label \| msg)$ and $M_{i+1} = \mathsf{HMAC}(k, M_i)$ for $i \in [1, n-1]$. For a bit-string x, $\mathsf{Trunc}_m(x)$ denotes truncating x to the leftest m bits.

Stateful AEAD scheme. The TLS protocol adopts a stateful AEAD scheme $(\mathsf{stE.Enc}, \mathsf{stE.Dec})$ to encrypt/decrypt messages in the handshake and record layers. The encryption algorithm $\mathsf{stE.Enc}(\mathsf{key}, \ell_{\mathrm{C}}, \mathrm{H}, \mathbf{M}, \mathsf{st}_e)$ takes as input a secret key key, a target ciphertext length ℓ_{C}, a header H, a message $\mathbf{M}$ and a state st_e, and outputs a ciphertext CT. The decryption algorithm $\mathsf{stE.Dec}(\mathsf{key}, \mathrm{H}, \mathrm{CT}, \mathsf{st}_d)$ takes as input key, a header H, a ciphertext CT and a state st_d, and outputs a plaintext $\mathbf{M}$ or a special symbol $\perp$ indicating that the ciphertext is invalid. When the AEAD scheme is instantiated by AES128_GCM, $\mathsf{stE.Enc}(\mathsf{key}, \ell_{\mathrm{C}}, \mathrm{H}, \mathbf{M}, \mathsf{st}_e)$ has the following steps:

1. Compute $\mathrm{Z}_0 := \mathsf{AES}(\mathsf{key}, \mathsf{st}_e)$ and update $\mathsf{st}_e := \mathsf{st}_e + 1$.

2. Suppose $\mathbf{M}$ is padded as $(\mathrm{M}_1, \ldots, \mathrm{M}_n)$ with $\mathrm{M}_i \in \{0,1\}^{128}$. From $i = 1$ to n, compute $\mathrm{Z}_i := \mathsf{AES}(\mathsf{key}, \mathsf{st}_e)$ and $\mathrm{C}_i := \mathrm{Z}_i \oplus \mathrm{M}_i$ and update $\mathsf{st}_e := \mathsf{st}_e + 1$. Set $\mathbf{C} := (\mathrm{C}_1, \ldots, \mathrm{C}_n)$.

3. Suppose that the header H has been padded as an element in $\mathbb{F}_{2^{128}}$. Let ℓ_{H} be the bit length of H. Given a vector $\mathbf{X} \in (\mathbb{F}_{2^{128}})^m$, the GHASH polynomial $\Phi_{\mathbf{X}} : \mathbb{F}_{2^{128}} \to \mathbb{F}_{2^{128}}$ is defined as $\Phi_{\mathbf{X}}(h) = \sum_{i=1}^m X_i \cdot h^{m-i+1} \in \mathbb{F}_{2^{128}}$. Compute $h := \mathsf{AES}(\mathsf{key}, \mathbf{0})$ and a GMAC tag $\sigma := \mathrm{Z}_0 \oplus \Phi_{(\mathrm{H}, \mathbf{C}, \ell_{\mathrm{H}}, \ell_{\mathrm{C}})}(h)$.

4. Output $\mathrm{CT} = (\mathbf{C}, \sigma)$.

Algorithm $\mathsf{stE.Dec}(\mathsf{key}, \mathrm{H}, \mathrm{CT}, \mathsf{st}_d)$ has the same steps as $\mathsf{stE.Enc}$, except for the following differences:

- Parse CT as $(\mathbf{C}, \sigma)$ and $\mathbf{C}$ as $(\mathrm{C}_1, \ldots, \mathrm{C}_n)$. Compute $\mathrm{M}_i := \mathrm{Z}_i \oplus \mathrm{C}_i$ for $i \in [1, n]$ and set $\mathbf{M} = (\mathrm{M}_1, \ldots, \mathrm{M}_n)$.
- Compute a tag σ' as described above and check $\sigma = \sigma'$.
- If the check passes, output $\mathbf{M}$. Otherwise, output $\perp$.

2.2 Security Model and Functionalities

We use the standard ideal/real security model (shown as below) to prove security of our protocol. We describe the definitions of ideal functionalities for Oblivious Transfer (OT) and OLEe in Section A. The functionalities for additively homomorphic commitments, standard commitments and Interactive Zero-Knowledge (IZK) proofs along with their instantiations can be found in the full version [65, Section A].

Ideal/real security model. We use the standard ideal/real paradigm [19, 32] to prove security of our protocol in the presence of a *malicious, static* adversary. In the *ideal-world* execution, the parties interact with a functionality $\mathcal{F}$, and some of them may be corrupted by an *ideal-world adversary* (a.k.a., *simulator*) $\mathcal{S}$. In the *real-world* execution, the parties interact with each other in an execution of protocol Π, and some of them may be corrupted by a *real-world adversary* $\mathcal{A}$ (that is often called an adversary for simplicity). We say that protocol Π securely realizes functionality $\mathcal{F}$, if the output of the honest parties and $\mathcal{A}$ in the real-world execution is computationally indistinguishable from the output of the honest parties and $\mathcal{S}$ in the ideal-world execution. We consider security with abort, and thus allow the ideal-world/real-world adversary to abort the functionality/protocol execution at some point. We prove security of our protocol in the $\mathcal{G}$-hybrid model in which the parties execute a protocol with real messages and also have access to a sub-functionality $\mathcal{G}$.

Use case. Similar to DECO [71], our protocol involves three parties: a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and a TLS server $\mathcal{S}$, where $\mathcal{P}$ and $\mathcal{V}$ jointly play the role of the client to interact with the server $\mathcal{S}$. Prover $\mathcal{P}$ has data stored on $\mathcal{S}$, and intends to prove to $\mathcal{V}$ about properties of the data, without any modification to the TLS server. For example, a Coinbase user (i.e., $\mathcal{P}$) wants to prove to a loan agency (i.e., $\mathcal{V}$) that his wallet balance satisfies an agreed-upon predicate (e.g., it is greater than a threshold).

Functionality $\mathcal{F}_{\mathsf{AuthData}}$

This functionality interacts with a prover $\mathcal{P}$, a verifier $\mathcal{V}$, a server $\mathcal{S}$ and an adversary.

- Upon receiving $(\mathsf{sid}, \mathsf{Query}, \alpha, \mathcal{S})$ from $\mathcal{P}$ and $(\mathsf{sid}, \mathsf{Query})$ from $\mathcal{V}$, where sid is a session identifier, Query is a query template and α is a private input for Query,
  1. Compute a query $Q := \mathsf{Query}(\alpha)$, and then send a pair (sid, Q) to $\mathcal{S}$.
  2. Receive a response (sid, R) from $\mathcal{S}$ and then store a tuple (sid, Q, R).
  3. Send $(\mathsf{sid}, |Q|, |R|, \mathcal{S})$ to the adversary.
- Upon receiving $(\mathsf{commit}, \mathsf{sid}, \mathsf{cid})$ from $\mathcal{P}$, where cid is a fresh commitment identifier, if a tuple (sid, Q, R) was previously stored, update it as $(\mathsf{sid}, \mathsf{cid}, Q, R)$, and send $(\mathsf{committed}, \mathsf{sid}, \mathsf{cid})$ to $\mathcal{V}$ and the adversary.
- Output $(\mathsf{sid}, \mathsf{cid}, Q, R)$ to $\mathcal{P}$ and $(\mathsf{sid}, \mathsf{cid}, \mathcal{S})$ to $\mathcal{V}$.

Figure 1: **Functionality for authenticating web data.**

Ideally, the user wants to prove it without revealing any other information (e.g., transaction details or the exact balance). Our protocol enables this: (1) $\mathcal{P}$ interacts with $\mathcal{S}$ via TLS to get the balance; (2) $\mathcal{P}$ can prove to $\mathcal{V}$ that the balance sent over the TLS protocol satisfies the predicate. The protocol ensures that $\mathcal{V}$ learns nothing except for that the predicate is true, and that $\mathcal{P}$ cannot use an inconsistent balance. We refer the reader to [71] for more application examples.

Functionality for authenticating web data. We model the security of authenticating web data by giving an ideal functionality. At a high level, the protocols to authenticate web data will involve the following steps performed in a secure and distributed way:

1. $\mathcal{P}$ and $\mathcal{V}$ (on behalf of the client) run the TLS protocol with $\mathcal{S}$ to establish an authenticated and confidential channel.
2. Under the secure channel, $\mathcal{P}$ sends a query Q to $\mathcal{S}$ and receives a response R from $\mathcal{S}$.
3. $\mathcal{P}$ sends the commitment of (Q,R) to $\mathcal{V}$, and convinces $\mathcal{V}$ that the commitment is correct on a valid pair (Q,R).
4. Given (Q,R) and its commitment, $\mathcal{P}$ can prove in zero knowledge to $\mathcal{V}$ that (Q,R) satisfies some statement.

In this paper, we focus on constructing a secure protocol to realize the first three steps. The final step can be realized using a variety of zero-knowledge proofs such as zk-SNARKs [10, 30, 33]. In this setting, the server $\mathcal{S}$ is always honest to run the protocol, [3] and so the security only needs to be guaranteed when either $\mathcal{P}$ or $\mathcal{V}$ is corrupted. For adversarial model, we consider a static, malicious adversary $\mathcal{A}$ who can corrupt one of $\mathcal{P}$ and $\mathcal{V}$ and may deviate the protocol arbitrarily. The ideal functionality for authenticating web data is defined in Figure 1, and builds upon the definition of the oracle functionality in [71]. Following an example in [71], a query template could be $\mathsf{Query}(\alpha) =$ "stock price of GOOG on June 1st, 2023 with API key = α".

Functionality $\mathcal{F}_{\mathsf{AuthData}}$ (shown in Figure 1) implies the following security properties, where similar properties were described in DECO [71].

- *Prover-integrity* : A malicious prover $\mathcal{P}$ cannot cause the query and response, whose commitments are sent to an honest verifier $\mathcal{V}$, to be inconsistent from that received or sent by the server $\mathcal{S}$.
- *Verifier-integrity* : A malicious verifier $\mathcal{V}$ cannot cause $\mathcal{P}$ to receive an incorrect response, i.e., if $\mathcal{P}$ outputs (Q,R), R must be $\mathcal{S}$'s response to the query Q sent by $\mathcal{P}$.
- *Privacy* : A malicious verifier $\mathcal{V}$ cannot learn any information on query Q and response R, except for the public information $(|Q|, |R|, \mathsf{Query})$ and which server $\mathcal{S}$ is accessed.

[3] We do not require any server-side modification or cooperation.

In the ideal world, all channels between honest parties and functionality $\mathcal{F}_{\mathsf{AuthData}}$ are confidential and authenticated. This guarantees the privacy of secret values Q,R. As in [71], we always consider that the length of a query $|Q|$, the length of a response $|R|$ and the name of a server $\mathcal{S}$ are known by the adversary. We use an identifier cid to represent a commitment on the query Q and response R. From the definition of $\mathcal{F}_{\mathsf{AuthData}}$, we have that the query-response pair (Q,R) committed by $\mathcal{F}_{\mathsf{AuthData}}$ are always consistent. The adversary who corrupts $\mathcal{P}$ can only get an identifier cid and has no way to tamper the values committed, which guarantees the prover-integrity. The honest prover $\mathcal{P}$ will always output a response R from $\mathcal{F}_{\mathsf{AuthData}}$, which is consistent with Q. Thus, the adversary who corrupts $\mathcal{V}$ cannot make the honest prover receive an inconsistent response, which guarantees the verifier-integrity.

3 Technical Overview

Third-party authentication of TLS payload could be achieved using a malicious 2PC protocol with a high overhead [71]. Our key technique is to first garble and evaluate circuits, and then prove the correctness of the resulting outputs in zero-knowledge. This enables us to use lightweight MPC building blocks, i.e., plain Two-Party Computation protocols based on Garbled Circuits (GC-2PC) that are the same as semi-honest protocols [56, 69, 70] except for using malicious OT instead of semi-honest OT, and the recent VOLE-based interactive zero-knowledge (IZK) proofs [7, 28, 63]. Our garble-then-prove technique can be used to authenticate web data for TLS, and may also be of independent interest for other applications in which all secrets are able to be known by a prover at the end, e.g., authenticating data from protocols like QUIC [24, 39], OAuth [35] and OpenID Connect [58].

We also present a technique to convert from IT-MACs to additively homomorphic commitments that are friendly to zk-SNARKs. This technique could also be used in other applications such as zero-knowledge machine learning [64]. Through the TLS application, we give an overview of these techniques. Furthermore, we provide several tailored optimizations to further improve the efficiency, based on the details of the TLS protocol. To help better understand our protocol, we first give a detailed overview of the TLS protocol.

3.1 An Overview of the TLS Protocol

In Figure 2, we provide a pictorial overview, and show complete details in the full version [65]. The protocol is executed between a TLS client ($\mathcal{C}$) and a TLS server ($\mathcal{S}$). It can be roughly divided into 4 phases:

- **Phase 1: pre-master secret.** $\mathcal{C}$ samples a random nonce $r_C \leftarrow \{0,1\}^{256}$, and then sends $\textsc{req}_C = r_C$ to $\mathcal{S}$. Then, $\mathcal{S}$ samples a random nonce $r_S \in \{0,1\}^{256}$, a random element $t_S \in \mathbb{Z}_q$, and computes a group element $T_S := t_S \cdot G$. $\mathcal{S}$ sends

Handshake Protocol

$r_C \leftarrow \{0,1\}^{256}$, $\text{REQ}_C := r_C$	$\xrightarrow{\text{REQ}_C}$	$r_S \leftarrow \{0,1\}^{256}, t_S \leftarrow \mathbb{Z}_q$
		$T_S := t_S \cdot G$
		$\sigma_S \leftarrow \mathsf{Sign}(\mathsf{sk}_S, r_C \| r_S \| T_S)$
Verify cert_S and σ_S	$\xleftarrow{\text{RES}_S}$	$\text{RES}_S := (r_S, T_S, \mathsf{cert}_S, \sigma_S)$
$t_C \leftarrow \mathbb{Z}_q$		
$T_C := t_C \cdot G$		
$\mathsf{pms} := \mathsf{F}_x(t_C \cdot T_S)$	$\xrightarrow{\text{RES}_C := T_C}$	$\mathsf{pms} := \mathsf{F}_x(t_S \cdot T_C)$

$$(\mathsf{ms}, \mathsf{key}_C, \mathsf{st}_C, \mathsf{key}_S, \mathsf{st}_S) \leftarrow \mathsf{Derive}(\mathsf{pms})$$

$\tau_C := \mathsf{H}(\text{REQ}_C, \text{RES}_S, \text{RES}_C)$		
$\text{UFIN}_C := \mathsf{PRF}_{96}(\mathsf{ms}, \mathsf{PublicStr} \| \tau_C)$		
$\text{FIN}_C \leftarrow \mathsf{stE.Enc}(\mathsf{key}_C, \text{UFIN}_C, \mathsf{st}_C)$	$\xrightarrow{(\text{H}_C, \text{FIN}_C)}$	$\text{UFIN}_C \leftarrow \mathsf{stE.Dec}(\mathsf{key}_C, \text{FIN}_C, \mathsf{st}_C)$
		Check UFIN_C
		$\tau_S := \mathsf{H}(\text{REQ}_C, \text{RES}_S, \text{RES}_C, \text{UFIN}_C)$
		$\text{UFIN}_S := \mathsf{PRF}_{96}(\mathsf{ms}, \mathsf{PublicStr} \| \tau_S)$
$\text{UFIN}_S \leftarrow \mathsf{stE.Dec}(\mathsf{key}_S, \text{FIN}_S, \mathsf{st}_S)$	$\xleftarrow{(\text{H}_S, \text{FIN}_S)}$	$\text{FIN}_S \leftarrow \mathsf{stE.Enc}(\mathsf{key}_S, \text{UFIN}_S, \mathsf{st}_S)$
Check UFIN_S		

Record Protocol

$\text{ENC}_Q \leftarrow \mathsf{stE.Enc}(\mathsf{key}_C, Q, \mathsf{st}_C)$	$\xrightarrow{(\text{H}_Q, \text{ENC}_Q)}$	$Q \leftarrow \mathsf{stE.Dec}(\mathsf{key}_C, \text{ENC}_Q, \mathsf{st}_C)$
$R \leftarrow \mathsf{stE.Dec}(\mathsf{key}_S, \text{ENC}_R, \mathsf{st}_S)$	$\xleftarrow{(\text{H}_R, \text{ENC}_R)}$	$\text{ENC}_R \leftarrow \mathsf{stE.Enc}(\mathsf{key}_S, R, \mathsf{st}_S)$

Figure 2: **Graphical depiction of TLS.** $\mathsf{PublicStr}$ refers to strings defined in the TLS specification. H_C, H_S, H_Q and H_R are public metadata headers defined by the TLS specification. Some details are omitted.

back $\text{RES}_S = (r_S, t_S, \mathsf{cert}_S, \sigma_S)$, where cert_S is a certification and σ_S is a signature on (r_C, r_S, T_S). To finish the key-exchange protocol, $\mathcal{C}$ sends back a random group element $T_C := t_C \cdot G$. Now both parties agree on a pre-master secret $\mathsf{pms} = \mathsf{F}_x(t_C \cdot T_S) = \mathsf{F}_x(t_S \cdot T_C)$, where F_x is a function mapping an elliptic-curve point to its x-coordinate.

- **Phase 2: TLS session keys.** With pms, $\mathcal{C}$ and $\mathcal{S}$ compute a master secret $\mathsf{ms} := \mathsf{PRF}_{384}(\mathsf{pms}, \text{“master secret”}, r_C \| r_S)$. Then, both parties compute a tuple $(\mathsf{key}_C, IV_C, \mathsf{key}_S, IV_S) := \mathsf{PRF}_{448}(\mathsf{ms}, \text{“key expansion”}, r_S \| r_C)$, where $\mathsf{key}_C, \mathsf{key}_S \in \{0,1\}^{128}$ are two application keys and $IV_C, IV_S \in \{0,1\}^{96}$ are the initial states $\mathsf{st}_C, \mathsf{st}_S$ of AEAD encryption. In Figure 2, we refer to the whole process as Derive.

- **Phase 3: Finished messages.** Two parties exchange test messages, which have already been known by them, over the established AEAD-encrypted channel. The client's message is $\text{UFIN}_C = \mathsf{PRF}_{96}(\mathsf{ms}, \text{“client finished”}, \tau_C)$, where τ_C is the hash of the TLS transcripts so far. $\mathcal{C}$ sends the AEAD ciphertext FIN_C of this message, which is encrypted with key_C and st_C, to $\mathcal{S}$. The server decrypts FIN_C and checks if UFIN_C is correct based on the same session key and values. Then $\mathcal{S}$ sends back a similarly encrypted message, and $\mathcal{C}$ checks its correctness.

- **Phase 4: Exchange payload.** Finally, two parties exchange their application payload. The exact process is essentially the same as Phase 3, with updated states for AEAD (based on AES-GCM), except that now the underlying payload is provided by the client and server based on the application. This phase could exchange several rounds of payload, depending on the application. In Figure 2, we only show one round of payload for simplicity. The following technical overview focuses on the one-round case, and our protocol can be extended to support multiple rounds of payload (see Section 4.2 for details).

3.2 Our Protocol Design

Now we introduce high-level ideas of our protocol based on the key observations described in Section 1.1. When describing our protocol, we use a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, who jointly emulate $\mathcal{C}$, the TLS client.

3.2.1 Phase 1: Generating pre-master secret

The process of generating pre-master secret pms in TLS is essentially a Diffie-Hellman (DH) key exchange. Since neither $\mathcal{P}$ nor $\mathcal{V}$ can know the outcome, they need to jointly emulate the TLS client. The first round of interaction of messages $(\text{REQ}_C, \text{RES}_S)$ can be done by $\mathcal{P}$ alone without $\mathcal{V}$. The message RES_C and DH secret needs to be distributively computed by $\mathcal{P}$ and $\mathcal{V}$. In more detail, $\mathcal{P}$ and $\mathcal{V}$ pick $t_{\mathcal{P}} \leftarrow \mathbb{Z}_q$, and $t_{\mathcal{V}} \leftarrow \mathbb{Z}_q$ respectively; $\mathcal{V}$ sends $t_{\mathcal{V}} \cdot G$ to $\mathcal{P}$, who defines $\text{RES}_C := (t_{\mathcal{P}} + t_{\mathcal{V}}) \cdot G$ and sends it to the server. In particular, $\mathcal{P}$ and $\mathcal{V}$ have an additive secret sharing (i.e., $t_{\mathcal{P}} \cdot T_S$ and $t_{\mathcal{V}} \cdot T_S$) of the DH secret $(t_{\mathcal{P}} + t_{\mathcal{V}}) \cdot T_S$. The above step is similar to the previous protocols [60, 71], who then use a fully secure multiplicative-to-additive conversion protocol, a.k.a, Oblivious Linear Evaluation (OLE), to convert an additive sharing of the EC point $(t_{\mathcal{P}} + t_{\mathcal{V}}) \cdot T_S$ to an additive sharing of its x-coordinate (i.e., pms).

Obtaining fully secure OLE is often expensive and requires tailored zero-knowledge proofs or excessive communication. However, in this particular setting, we show that an OLE with error (OLEe), where the error could even depend on parties' inputs, is already sufficient. Such an OLEe can be efficiently computed using $\log q$ correlated OTs without the need of any extra checks. This would lead to one-bit information leakage about pms to the adversary who corrupts the prover $\mathcal{P}$. However, due to the TLS protocol, pms is of high entropy and we can show that such leakage does not help the adversary in guessing the whole secret pms. Intuitively, such an error could only lead to the selective-failure attack, which allows the adversary to guess c bits of the secret with probability 2^{-c}, but if the guess is incorrect the protocol execution aborts. Such an attack does not reduce concrete security since the adversary could bet on c bits of the secret too. A similar analysis has already been used in designing maliciously secure protocols (e.g., [22, 44, 67]).

3.2.2 Phase 2: Deriving TLS session keys

Now $\mathcal{P}$ and $\mathcal{V}$ hold an additive secret sharing of pms and need to derive additive sharings of TLS session keys using PRF based on HMAC-SHA256. This is the most expensive part for TLS handshake in DECO, who implemented this step using a fully malicious 2PC protocol to compute a circuit containing 779,213 AND gates. We show how to achieve a $16\times$ improvement in communication.

Eliminating malicious 2PC via garble-then-prove. We observe that using a fully malicious 2PC is a complete overkill for applications that allow a verifier to reveal all its secrets to a prover later (e.g., authenticating web data for TLS). In our protocol, we use a plain GC-2PC protocol with malicious OT between $\mathcal{P}$ and $\mathcal{V}$ to jointly derive session keys. In more detail, $\mathcal{P}$ is the circuit garbler and $\mathcal{V}$ is the circuit evaluator. Any value that needs to be revealed to both parties is revealed to $\mathcal{V}$ first (by letting $\mathcal{P}$ send the decoding information to $\mathcal{V}$), who sends back the value to $\mathcal{P}$. In this way, $\mathcal{V}$ cannot break the privacy requirement of the function being computed (but can still change the output, which can be detected later). However, a malicious $\mathcal{P}$ can cheat in a seemingly catastrophic way: a malicious $\mathcal{P}$ could change a Garbled Circuit (GC) to control the output to be anything (could even be pms or something that can help $\mathcal{P}$ recover pms).

As we discussed in the main philosophy, instead of preventing $\mathcal{P}$ from cheating, we ensure that $\mathcal{P}$'s cheating behavior can be caught by $\mathcal{V}$ in hindsight. In more detail, we ask $\mathcal{P}$ to also commit to $\mathcal{V}$ its input, i.e., $\mathcal{P}$'s share of pms. Since we reveal the value to two parties by $\mathcal{V}$ getting it first, $\mathcal{P}$'s cheating behavior is "well-defined": $\mathcal{V}$ has its own share of pms, the commitment of the other share of pms, and the output of the GC that $\mathcal{P}$ garbled. If we later reveal $\mathcal{V}$'s secret to $\mathcal{P}$ after the TLS protocol terminates, $\mathcal{P}$ has all secrets (in particular, $\mathcal{P}$ knows $\mathcal{V}$'s share of pms) and can use a ZK protocol to prove that all outputs obtained by $\mathcal{V}$ are correct. We emphasize that $\mathcal{P}$ does not prove the correctness of the GC, and thus we are using GC in a black-box way. In conclusion, although $\mathcal{V}$ does not have a guarantee on $\mathcal{P}$'s honesty during the protocol execution, $\mathcal{V}$ can detect any cheating in hindsight as long as the GC output is first revealed to $\mathcal{V}$.

This optimization alone significantly reduces the overhead of the protocol as it eliminates the need of a malicious 2PC protocol, which is expensive in computation/communication but also requires memory linear to the circuit size to store the preprocessing triples. We formally model the 2PC with garble-then-prove approach as an ideal functionality $\mathcal{F}_{\mathsf{GP2PC}}$ shown in the full version [65, Section 4.1], and show how to instantiate $\mathcal{F}_{\mathsf{GP2PC}}$ using plain GC-2PC with malicious OT and interactive ZK, which is described in the full version [65, Section 4.2]. The 2PC protocol with garble-then-prove approach may be of independent interest, and may be applied in other scenarios such that all $\mathcal{V}$'s secrets are allowed to be revealed to $\mathcal{P}$ at the end.

TLS-specific circuit optimization. Our second optimization is to minimize the circuit to be computed in the protocol above. By using unique features of how session keys are derived in TLS, we are able to reduce the circuit size from 779,213 to 289,827 AND gates, a $2.7\times$ improvement. Let's look at master secret ms as an example, which has a 384-bit output. The exact derivation formula is as follows:

$$V = \text{"master secret"} \| r_C \| r_S \in \{0,1\}^{592},$$
$$M_1 = \mathsf{HMAC}(\mathsf{pms}, V) \in \{0,1\}^{256},$$
$$M_2 = \mathsf{HMAC}(\mathsf{pms}, M_1) \in \{0,1\}^{256},$$
$$\mathsf{ms} = \mathsf{HMAC}(\mathsf{pms}, M_1 \| V) \| \mathsf{Trunc}_{128}\big(\mathsf{HMAC}(\mathsf{pms}, M_2 \| V)\big).$$

In the above equation, $\mathsf{HMAC}(k,m) = \mathsf{SHA256}(k \oplus \text{opad}, \mathsf{SHA256}(k \oplus \text{ipad}, m))$, and that $\mathsf{SHA256}(m_1, m_2, m_3) = f_{\mathsf{H}}(f_{\mathsf{H}}(f_{\mathsf{H}}(IV_0, m_1), m_2), m_3)$ where m_i's are 512-bit strings.

To compute an HMAC-SHA256, we need at least 4 SHA256 compress calls: 2 calls to compute the outer hash and at least 2 calls to compute the inner hash; if m is longer than 447 bits, the inner hash requires even more calls.

Although there are totally 19 SHA256 compression calls to derive ms, we found that only 6 of them need to be computed in GC-2PC. First, $IV_1 = f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{ipad})$ and $IV_2 = f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{opad})$ only need to be computed once in GC-2PC and they can be kept as garbled labels to be reused in all HMAC computation. Second, the messages to all HMAC are public, which can be used for optimization: we reveal the value IV_1 while keeping IV_2 secret, so that subsequent computation taking IV_1 and the message can be done locally. We show the exact computation as follows:

$$
\begin{aligned}
M_1 &= f_{\mathsf{H}}\big(f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{opad}), f_{\mathsf{H}}(f_{\mathsf{H}}(f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{ipad}), \\
&\quad m_1), m_2)\big) \\
M_2 &= f_{\mathsf{H}}\big(f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{opad}), f_{\mathsf{H}}(f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{ipad}), M_1)\big) \\
\mathsf{ms} &= f_{\mathsf{H}}\big(f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{opad}), f_{\mathsf{H}}(f_{\mathsf{H}}(f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{ipad}), \\
&\quad M_1 \| V_1), V_2)\big) \| \mathsf{Trunc}_{128}\big(f_{\mathsf{H}}\big(f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{opad}), \\
&\quad f_{\mathsf{H}}(f_{\mathsf{H}}(f_{\mathsf{H}}(IV_0, \mathsf{pms} \oplus \mathsf{ipad}), M_2 \| V_1), V_2)\big)\big),
\end{aligned}
$$

where red refers to computation in GC-2PC, green refers to local computation, and blue refers to reused values. In the above equations, (m_1, m_2) and (V_1, V_2) are the bit-strings about V when suitably padding V to specific bits. The process of deriving $(\mathsf{key}_C, IV_C, \mathsf{key}_S, IV_S)$ is very similar to the above and also takes 6 SHA256 compression calls. Later, computing UFIN_C takes another 2 compression calls in GC-2PC. As a result, the whole circuit computing all needed HMAC takes 289,827 AND gates. This optimization is secure in the random oracle model (see the full version [65, Section E] for details).

3.2.3 Phase 3: Finished messages

Using a similar protocol, we compute (UFIN_C, UFIN_S) and reveal them to both parties. [4] Now the main task is to perform AEAD encryption/decryption on public plaintext/ciphertext and secretly shared AEAD keys. Our focus in this paper is AES-GCM (see Section 2.1 for a quick recall of the scheme), which is the main scheme used over the Internet right now. We take distributedly performing AEAD encryption as an example, and performing AEAD decryption is totally similar. Note that DECO mainly supports CBC-HMAC and could support AES-GCM by computing in a Boolean circuit all ciphertext blocks C_i's and powers h^i's using a fully malicious 2PC, where $\text{C}_i = \mathsf{AES}(\mathsf{key}, \mathsf{st} + i) \oplus \text{M}_i$ for a state st and a plaintext M_i, $h = \mathsf{AES}(\mathsf{key}, \mathbf{0})$ and $\mathsf{key} \in \{\mathsf{key}_C, \mathsf{key}_S\}$ is an application key. By revealing C_i to both parties while only revealing an additive sharing of h^i, $\mathcal{P}$ and $\mathcal{V}$ can compute an additive sharing of the GMAC tag locally. This method can be very costly since it requires securely computing a number of finite field multiplications equal to the number of AES calls. What's more, the circuit to compute a multiplication over $\mathbb{F}_{2^{128}}$ has at least 8,765 AND gates, even larger than the AES circuit itself!

AES-GCM computation consists of two tasks: computing the ciphertext and computing the GMAC tag. The first task is relatively easy as we can use the garble-then-prove approach again to avoid malicious 2PC, where the plaintext is known by both parties in this phase. However, computing the GMAC tag is more complicated. Roughly speaking, the GMAC tag is an inner product between a public vector over $\mathbb{F}_{2^{128}}$ and a private vector $(\text{z}_0, h^1, \ldots, h^n)$ where $\text{z}_0 = \mathsf{AES}(\mathsf{key}, \mathsf{st})$ is shared by both parties. Revealing any term in the second vector would allow the adversary to forge a GMAC tag on any message of its choice. Computing z_0 can be done in GC-2PC; however, since we reveal the additive shares of z_0, meaning that the output is not well defined from $\mathcal{V}$'s perspective, the garble-then-prove approach does not immediately work. To solve this issue, we ask $\mathcal{P}$ to commit to its share of z_0. After the completion of the TLS protocol, when $\mathcal{P}$ knows all secrets, $\mathcal{P}$ will prove the computation with respect to the above commitment. To avoid computing h^i in circuits, we also reveal the additive shares of h together with z_0. Then two parties use an OLEe over $\mathbb{F}_{2^{128}}$ to compute additive sharings on all powers of h. This way, each term only needs *2KB* communication, 100× smaller than computing in GC-2PC! Similar to the use of OLEe in phase 1, this also introduces a chance of a selective failure attack; however, it can be easily shown that providing multiple chances of selective failure attacks does not provide any more power to the adversary.

3.2.4 Phase 4: Payload

This phase is the first time $\mathcal{P}$ provides a private input (namely the query string) that is not part of the TLS execution. The overall protocol is similar to phase 3 how we compute the finished messages, except that the plaintext to AES-GCM-based AEAD is not public anymore. Therefore, we can mostly follow the phase-3 protocol except that $\mathcal{P}$ XOR its query to the additive share of AES output, and then sends the resulting value to $\mathcal{V}$. In this way, $\mathcal{V}$ can obtain the ciphertext directly by XORing the resulting value with its additive share.

After obtaining the AEAD cihertexts ENC_Q and ENC_R on the query Q and response R from $\mathcal{P}$, $\mathcal{V}$ opens $t_{\mathcal{V}} \in \mathbb{Z}_q$ to $\mathcal{P}$, who can replay the whole TLS protocol to obtain all values computed in GC-2PC. At this point, $\mathcal{V}$ holds 1) the commitment to $\mathcal{P}$'s share of pms; 2) the commitments to all values revealed from GC-2PC as XOR shares of AES outputs; 3) the values revealed from GC-2PC to both parties. Now $\mathcal{P}$ can prove to $\mathcal{V}$ in zero-knowledge that the whole computation is correct with respect to the commitments and values that $\mathcal{V}$ has. The circuit proven in ZK includes 1) the circuit computed

[4] We could postpone the verification of UFIN_S to the phase in which $\mathcal{P}$ obtains all secrets and then can *locally* check UFIN_S. This optimization removes the GC-2PC to compute UFIN_S (see Section 4.2 for more details).

in GC-2PC and 2) the decryption of the ciphertext to the response. However, the cost of ZK is significantly smaller than GC-2PC: when using the latest VOLE-based ZK [66], the communication of ZK is only 1 bit per AND gate, compared to 256 bits per AND gate required by the GC-2PC protocol [70]. During the process of ZK, $\mathcal{P}$ also needs to commit to the plaintext of the query and response to prove AEAD computation. They will be converted to a ZK-friendly format in the next phase.

3.2.5 Converting to ZK-Friendly Commitments

Now $\mathcal{V}$ has commitments of the query Q and response R that $\mathcal{P}$ knows. Their correctness has been verified by $\mathcal{V}$ through VOLE-based ZK. Such commitments are instantiated by IT-MACs and denoted by $[\![\boldsymbol{u}]\!] = ([\![u_1]\!], \ldots, [\![u_\ell]\!])$, where for each $i \in [1, \ell]$, $u_i \in \{0,1\}$, $(u_i, \mathsf{M}[u_i])$ is obtained by $\mathcal{P}$, $(\mathsf{K}[u_i], \Delta)$ is held by $\mathcal{V}$, and $\mathsf{M}[u_i] = \mathsf{K}[u_i] \oplus u_i \Delta$.

We first convert the IT-MACs from binary field $\mathbb{F}_{2^\lambda}$ to a large field $\mathbb{Z}_q$ for a prime q. Let $\mathsf{H} : \{0,1\}^\lambda \to \mathbb{Z}_q$ be a random oracle. For each component u_i, $\mathcal{V}$ computes $\widetilde{\mathsf{K}}[u_i] := \mathsf{H}(\mathsf{K}[u_i])$ and sends $W_i := \mathsf{H}(\mathsf{K}[u_i]) - \mathsf{H}(\mathsf{K}[u_i] \oplus \Delta) + \Gamma \in \mathbb{Z}_q$ to $\mathcal{P}$, who computes $\widetilde{\mathsf{M}}[u_i] := \mathsf{H}(\mathsf{M}[u_i]) + u_i \cdot W_i = \widetilde{\mathsf{K}}[u_i] + u_i \cdot \Gamma$, where $\Gamma \in \mathbb{Z}_q$ is a uniform global key known to $\mathcal{V}$. We also ask $\mathcal{P}$ to commit to (Q, R) using an additively homomorphic commitment (e.g., Pedersen [54] and KZG [42]) that is friendly to zkSNARKs. To check consistency between IT-MACs over $\mathbb{Z}_q$ and additively homomorphic commitments, we reveal a random linear combination of the values committed in two formats, where the random challenges are chosen by $\mathcal{V}$.

There are several extra considerations. First, the random linear combination would lead to some leakage, so both parties need to generate one more random value committed in both formats to mask the linear combination before it is revealed. Two commitments of the random value only need to be consistent in the honest case. Second, the values $\{W_i\}$ may *not* be computed correctly and thus after $\mathcal{P}$ opens the linear combination, $\mathcal{V}$ needs to open the values $\{W_i\}$ by revealing Δ and Γ, so that $\mathcal{P}$ can check that all values are computed correctly. Finally, the final check does not need to be done over bits but any packing of the values. This could significantly reduce the number of additively homomorphic commitments. In addition, two additional checks need to be performed to prevent the possible privacy leakage on $\boldsymbol{u}$ by using inconsistent Δ and Γ. See the full version [65, Section C] for details.

3.3 Protocol Summary

Previous discussions provide a high-level intuition on how we design the protocol. However, partially due to the complexity of TLS, the whole protocol is very complicated. Below, we provide a summary of the whole protocol omitting the details when considering the optimization shown in Footnote 4. The exact details of our protocol, along with the proof of security, can be found in Section 4 and related appendices.

1. $\mathcal{P}$ samples and sends REQ_C to $\mathcal{S}$ and gets back RES_S.
2. $\mathcal{P}$ forwards $(\text{REQ}_C, \text{RES}_S)$ to $\mathcal{V}$, who sends $t_\mathcal{V} \cdot G$ to $\mathcal{P}$. Then $\mathcal{P}$ picks $t_\mathcal{P}$ and sends $(t_\mathcal{P} + t_\mathcal{V}) \cdot G$ to $\mathcal{S}$. Then $\mathcal{P}$ and $\mathcal{V}$ run the conversion from an elliptic-curve point to its x-coordinate, based on OLE with errors, so that two parties obtain an additive sharing of pms.
3. Two parties run the GC-2PC protocol with the garble-then-prove technique to derive the key material and client finished message. In particular, they obtain: 1) XOR shares of $h_C = \mathsf{AES}(\mathsf{key}_C, \mathbf{0})$ and $z_C = \mathsf{AES}(\mathsf{key}_C, \mathsf{st}_C)$; 2) initial vectors IV_C, IV_S, intermediate public values revealed in HMAC-based PRF, UFIN_C and its AES encryption; 3) $\mathsf{pms}, \mathsf{ms}, \mathsf{key}_S, \mathsf{key}_C$ in the form of garbled labels.
4. Based on OLEe over $\mathbb{F}_{2^{128}}$, $\mathcal{P}$ and $\mathcal{V}$ compute the GMAC tag with these XOR shares on (h_C, z_C) in the offline-online mode. Then $\mathcal{P}$ assembles FIN_C and sends it to the server $\mathcal{S}$. After receiving FIN_S from $\mathcal{S}$, $\mathcal{P}$ forwards it to $\mathcal{V}$.
5. $\mathcal{P}$ and $\mathcal{V}$ execute the GC-2PC protocol with the garble-then-prove approach to generate the AES ciphertext that encrypts $\mathcal{P}$'s query and XOR shares of an AES output $z_Q = \mathsf{AES}(\mathsf{key}_C, \mathsf{st}_C + 2)$. Both parties use OLEe and the XOR shares on (h_C, z_Q) to compute the GMAC tag, and then $\mathcal{P}$ sends the AEAD ciphertext ENC_Q on the query Q to $\mathcal{S}$. Then, $\mathcal{S}$ returns the AEAD ciphertext ENC_R on the response R to $\mathcal{P}$, who forwards it to $\mathcal{V}$.
6. $\mathcal{P}$ reveals its XOR shares of h_C, z_C, z_Q to $\mathcal{V}$, who can recover h_C, z_C, z_Q and then use them to *locally* verify the correctness of AEAD ciphertexts FIN_C and ENC_Q. Besides, ENC_R can also be locally verified by revealing the corresponding AES outputs to $\mathcal{V}$.
7. $\mathcal{V}$ sends $t_\mathcal{V}$ to $\mathcal{P}$ who checks that it is consistent with $t_\mathcal{V} \cdot G$ received earlier. $\mathcal{P}$ then computes $t_\mathcal{P} + t_\mathcal{V}$, and recovers all values in the execution of TLS, including all values revealed previously. If any value is incorrect, $\mathcal{P}$ aborts.
8. $\mathcal{V}$ now holds commitments to $\mathcal{P}$'s share of pms and values revealed as XOR shares earlier. $\mathcal{P}$ proves to $\mathcal{V}$ in zero-knowledge that these commitments are consistent with the values revealed to $\mathcal{V}$ based on the TLS specification.
9. Two parties run a protocol to convert the commitments on Q and R based on IT-MACs to additively homomorphic commitments like Pedersen on the same values.

4 Authenticating Web Data for TLS

In the full version [65, Section 4], we define an ideal functionality $\mathcal{F}_{\mathsf{GP2PC}}$ for 2PC in the garble-then-prove framework, and then show an efficient protocol securely realizing $\mathcal{F}_{\mathsf{GP2PC}}$ by using a plain GC-2PC protocol with malicious OT and the recent interactive ZK proof based on IT-MACs. Based on $\mathcal{F}_{\mathsf{GP2PC}}$ we provide a complete description of our protocol (denoted by Π_{AuthData}) that authenticates web data for TLS 1.2 in Section 4.1. Then, we show how to extend protocol Π_{AuthData} to support multi-round query-response sessions and describe further optimizations in Section 4.2. While Π_{AuthData} focuses on the case of reading user data, we also extend it to support for writing user data in Section 4.2. In Section 4.3, we also show how to extend our protocol to support TLS 1.3.

4.1 Detailed Protocol for Authenticating Data

Our protocol Π_{AuthData} is divided into four phases, where the last three phases are jointly called online phase.

- **Preprocessing:** A prover $\mathcal{P}$ and a verifier $\mathcal{V}$ generate correlated randomness before the TLS connection.
- **Handshake:** $\mathcal{P}$ and $\mathcal{V}$ call $\mathcal{F}_{\mathsf{GP2PC}}$ to perform client operations. This phase establishes the connection with $\mathcal{S}$ while neither of $\mathcal{P}$ and $\mathcal{V}$ know any secrets or application keys.
- **Record:** $\mathcal{P}$ and $\mathcal{V}$ call $\mathcal{F}_{\mathsf{GP2PC}}$ to encrypt a query Q, and then $\mathcal{P}$ locally decrypts the ciphertext on a response R.
- **Post-record:** In this phase, the TLS protocol has terminated. Now, $\mathcal{P}$ is allowed to know all secret values in the TLS session. Then $\mathcal{P}$ and $\mathcal{V}$ call $\mathcal{F}_{\mathsf{GP2PC}}$ to prove the correctness of all values revealed to $\mathcal{V}$. Finally, both parties transform IT-MACs of Q and R into their additive-homomorphic commitments, which are connected to a variety of zk-SNARKs.

Π_{AuthData} invokes the following three sub-protocols, whose details are described in the full version [65, Section B.4].

- Sub-protocol Π_{E2F} (shown in the full version [65, Section B.1]) converts additive sharings of elliptic-curve points into that of x-coordinates, and will be used to generate an additive sharing $[\mathsf{pms}]_p$ of pre-master secret.
- Sub-protocol Π_{PRF} (shown in the full version [65, Section B.2]) calls $\mathcal{F}_{\mathsf{GP2PC}}$ to compute HMAC-based PRF in the handshake phase. Then, it proves correctness of all opened values by calling $\mathcal{F}_{\mathsf{GP2PC}}$ in the post-record phase. Protocol Π_{PRF} will be used to generate the master secret ms, application keys $\mathsf{key}_C, \mathsf{key}_S$, initial vectors IV_C, IV_S and $\text{UFIN}_C, \text{UFIN}_S$.
- Sub-protocol Π_{AEAD} (shown in the full version [65, Section B.3]) calls $\mathcal{F}_{\mathsf{GP2PC}}$ to compute AES blocks used for encryption/decryption of AEAD, and uses OLEe to compute GMAC tags, in the handshake and record phases. In the post-record phase, Π_{AEAD} calls $\mathcal{F}_{\mathsf{GP2PC}}$ to prove correctness of all AES blocks, and invokes $\mathcal{F}_{\mathsf{IZK}}$ to generate IT-MACs $[\![Q]\!]$ on a query Q. Sub-protocol Π_{AEAD} is used to encrypt UFIN_C, Q to obtain the ciphertexts $\text{FIN}_C, \text{ENC}_Q$ and decrypt FIN_S to get UFIN_S.

$\mathcal{P}$ and $\mathcal{V}$ generate authenticated bits $[\![Q]\!]$ and $[\![R]\!]$ in the post-record phase by calling an ideal functionality $\mathcal{F}_{\mathsf{IZK}}$ for ZK proofs based on IT-MACs. Functionality $\mathcal{F}_{\mathsf{IZK}}$ (shown in the full version [65, Section A.4]) is a simple extension of the ideal functionality defined in [64], and can be securely realized using the recent VOLE-based ZK protocols [7,28,63,66]. Besides, $\mathcal{P}$ and $\mathcal{V}$ call an ideal functionality $\mathcal{F}_{\mathsf{Conv}}$ (shown in the full version [65, Section C]) to convert $[\![Q]\!]$ and $[\![R]\!]$ into their additively homomorphic commitments in the post-record phase. In the full version [65, Section C], we present an efficient protocol to securely realize $\mathcal{F}_{\mathsf{Conv}}$.

We postpone the details of protocol Π_{AuthData} to the full version [65, Section B.4]. As in DECO [71], protocol Π_{AuthData} focuses on the case of one-round query-response session, i.e., a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ jointly generate and send the AEAD ciphertext of *a single* query Q to a server $\mathcal{S}$ who returns the AEAD ciphertext of *a single* response R to $\mathcal{P}$. Note that one-round session is enough for a lot of applications [71]. For one-round session, $\mathcal{P}$ is *unnecessary* to decrypt the AEAD ciphertext ENC_R on the response R and verify its GMAC tag by running sub-protocol Π_{AEAD} with $\mathcal{V}$. These operations can be performed *locally* by $\mathcal{P}$ after it knows the server-to-client key key_S, where the TLS session terminates after ENC_R was received by $\mathcal{P}$ and forwarded to $\mathcal{V}$. Nevertheless, $\mathcal{P}$ and $\mathcal{V}$ still need to generate $[\![\mathsf{z}_R]\!]$ and $[\![R]\!]$ by calling functionality $\mathcal{F}_{\mathsf{IZK}}$. $\mathcal{V}$ also needs to check the correctness of the GMAC tag in ciphertext ENC_R via getting $h_S = \mathsf{AES}(\mathsf{key}_S, \mathbf{0})$ and $\mathsf{z}_R = \mathsf{AES}(\mathsf{key}_S, \mathsf{st}_d^C)$.

The security of protocol Π_{AuthData} depends on the PRF-Oracle-Diffie-Hellman (PRF-ODH) assumption, which has been used for proving the security of TLS 1.2 [40,48] and is recalled in the full version [65, Section E]. Besides, we assume that the underlying signature scheme satisfies Existential UnForgeability under Chosen-Message Attack (EUF-CMA).

Theorem 1. *If the PRF-ODH assumption holds and the underlying signature scheme is EUF-CMA secure, then protocol* Π_{AuthData} *securely realizes functionality* $\mathcal{F}_{\mathsf{AuthData}}$ *in the* $(\mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{IZK}}, \mathcal{F}_{\mathsf{Conv}})$*-hybrid model, assuming that the compression function* f_{H} *underlying* PRF *is a random oracle and* AES *is an ideal cipher.*

We provide a formal proof of Theorem 1 in the full version [65, Section E].

4.2 Extensions and Optimizations

Extend to multi-round query-response sessions. We are able to extend the protocol Π_{AuthData} to support multiple rounds of payload. Specifically, $\mathcal{P}$ and $\mathcal{V}$ can execute sub-protocol Π_{AEAD} (shown in the full version [65, Section B.3]) multiple times to encrypt multiple queries, where the additive sharings of powers of $h_C = \mathsf{AES}(\mathsf{key}_C, \mathbf{0})$ need to be computed only once and are reused among these sub-protocol executions. Note that the state st_C is always increased for computing multiple AEAD ciphertexts following the TLS specification. This prevents $\mathcal{P}$ or $\mathcal{V}$ to forge GMAC tags by using the same state for different ciphertexts.

If every query is independent of previous responses (Case 1), $\mathcal{P}$ can *locally* decrypt the AEAD ciphertexts of all responses, after the TLS session terminates and it obtains the server-to-client application key key_S. If every query relies on previous responses (Case 2), $\mathcal{P}$ has to decrypt the ciphertexts of all responses via interacting with $\mathcal{V}$. This can be done by running sub-protocol Π_{AEAD} with $\mathsf{type}_1 =$ "decryption" and $\mathsf{type}_2 =$ "secret", where Π_{AEAD} was designed for supporting decryption of AEAD ciphertexts in the record phase. During the protocol execution, Π_{AEAD} also allows $\mathcal{P}$ and $\mathcal{V}$ to verify the correctness of GMAC tags in AEAD ciphertexts of responses. Therefore, in both cases, $\mathcal{P}$ can check the correctness of AEAD ciphertext on every response via sending the ciphertext to $\mathcal{V}$ and then running sub-protocol Π_{AEAD} with $\mathcal{V}$, before generating the ciphertext on the next query. In fact, this is unnecessary and the GMAC tags of AEAD ciphertexts on all responses can be verified *locally* by $\mathcal{P}$ after it knows key_S (see below for discussion of this optimization). In Case 2, the decryption of the response's ciphertext sent in the final-round session can still be performed *locally* with key_S.

In the case of multi-round sessions, both $\mathcal{P}$ and $\mathcal{V}$ can use the same approach implied in the post-record phase of main protocol Π_{AuthData} (shown in the full version [65, Section B.4]) to check the correctness of all AEAD ciphertexts on multiple queries and responses. In Case 2, we note that $\mathcal{P}$ needs to decrypt the AEAD ciphertexts on responses using key_S, and then compares the resulting plaintexts with that obtained during the execution of sub-protocol Π_{AEAD}, when it performs the local verification with key_S in the post-record phase. This verification aims to check that no error is introduced to the responses computed via the distributed decryption in sub-protocol Π_{AEAD}. Both parties are also able to obtain the IT-MACs on all responses in a way totally similar to main protocol Π_{AuthData}. Note that the IT-MACs on all queries have already been obtained during multiple executions of sub-protocol Π_{AEAD}.

Optimization. We can further optimize the efficiency of protocol Π_{AuthData} by delaying the check of $\textsc{ufin}_S$ and AEAD ciphertext $\textsc{fin}_S$ from the handshake phase to the post-record phase. That is, $\mathcal{P}$ and $\mathcal{V}$ do *not* execute sub-protocol Π_{AEAD} to generate $\textsc{ufin}_S$ and the GMAC tag used to check $\textsc{fin}_S$. Instead, $\mathcal{P}$ can *locally* check their correctness after it obtains master secret ms. Verifier $\mathcal{V}$ checks the correctness of $\textsc{ufin}_S$ by calling the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ with $\mathcal{P}$, and then checks the correctness of $\textsc{fin}_S = (\textsc{c}, \sigma)$ in the following two steps:

1. $\mathcal{P}$ sends $\textsc{z}_1$ to $\mathcal{V}$, and then proves $\textsc{z}_1 = \mathsf{AES}(\mathsf{key}_S^*, IV_S + 1)$ by calling the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$, where key_S^* and IV_S are the server-to-client application key and initial vector whose correctness has been proved in the post-record phase. Then, $\mathcal{V}$ checks that $\textsc{ufin}_S \oplus \textsc{z}_1 = \textsc{c}$.
2. $\mathcal{V}$ checks the correctness of GMAC tag σ in the way shown in the post-record phase of protocol Π_{AuthData}.

This has *no* impact on privacy and integrity, as this optimization only delays the check. If one of these values is incorrect, $\mathcal{P}$ or $\mathcal{V}$ aborts. Note that this optimization is supported by the TLS implementation. Furthermore, this optimization can be applied in the case of multi-round sessions. That is, $\mathcal{P}$ and $\mathcal{V}$ can delay the correctness check of all AEAD ciphertexts on responses from the record phase into the post-record phase by checking the correctness of GMAC tags via the approach shown in main protocol Π_{AuthData}. Recall that $\mathcal{P}$ also checks that the responses output by sub-protocol Π_{AEAD} are identical to that via the local decryption with key_S, when it performs the local verification in the post-record phase. This optimization allows us to reduce communication rounds and improve the whole performance.

Extend to support for writing user data. Our protocol Π_{AuthData} focuses on reading user data from a website acting as the TLS server. For most of Web2 and Web3 applications, it is sufficient. Nevertheless, for a few applications, a user (i.e., prover) may be desirable to write its data on the website (e.g., updating personal information) during the protocol execution of Π_{AuthData}. In this case, a malicious verifier $\mathcal{V}$ may tamper the queries sent from a prover $\mathcal{P}$ to the TLS server by adding some errors into the AES ciphertexts on queries. The attack would be detected by $\mathcal{P}$ after it obtains all secrets. However, the user data on the website has already been tampered. To prevent such attacks, we need to extend functionality $\mathcal{F}_{\mathsf{GP2PC}}$ to an ideal functionality $\mathcal{F}_{\mathsf{GP2PC}}^{\mathsf{noerr}}$ (defined in the full version [65, Section 4.1]) that does not allow $\mathcal{V}$ to introduce any errors. In the full version [65, Section 4.2], we show how to extend the protocol instantiating $\mathcal{F}_{\mathsf{GP2PC}}$ to securely realize $\mathcal{F}_{\mathsf{GP2PC}}^{\mathsf{noerr}}$ with no extra communication. When replacing $\mathcal{F}_{\mathsf{GP2PC}}$ with $\mathcal{F}_{\mathsf{GP2PC}}^{\mathsf{noerr}}$, protocol Π_{AuthData} would allow $\mathcal{P}$ to securely write user data.

This holds for multi-round sessions. For the multi-round session extension as described above, we point out a caveat. If the writing queries relying on previous responses, then $\mathcal{P}$ and $\mathcal{V}$ need to execute sub-protocol Π_{AEAD} to check the correctness of the AEAD ciphertext on each response except for the final response, before sending the AEAD ciphertext on the next query to the server. Otherwise, the above optimization, which locally checks the AEAD ciphertexts on all responses after obtaining key_S, can still be used.

4.3 Extending Our Protocol for TLS 1.3

While the protocol Π_{AuthData} (shown in the full version [65, Section B.4]) focuses on the case of TLS 1.2, we are also able to extend it for TLS 1.3, and will implement the protocol to authenticate web data for TLS 1.3 in the future work. The main differences between TLS 1.2 and TLS 1.3 are the key derivation function (KDF) and the handshake phase. In this section, we focus on the handshake mode of full 1-RTT, where the optional mode of 0-RTT based on a pre-shared key can be securely computed in a similar way.

The key derivation in TLS 1.3 adopts the HMAC-based key derivation function (HKDF) [46, 47], which consists of two sub-functions: $\mathsf{HKDF.Extract}$ and $\mathsf{HKDF.Expand}$. Specifically, $\mathsf{prk} \leftarrow \mathsf{HKDF.Extract}(salt, \mathsf{ikm})$ takes as input a non-secret random *salt* and a secret input key material ikm, and then extracts a pseudorandom key prk, i.e., $\mathsf{prk} = \mathsf{HMAC}(salt, \mathsf{ikm})$. Note that *salt* is the HMAC key, and ikm is the HMAC input. In this case, we can securely compute $\mathsf{HKDF.Extract}$ in the following manner:

$$\mathsf{prk} = f_{\mathsf{H}}\big(f_{\mathsf{H}}(IV_0, salt \oplus \text{opad}), f_{\mathsf{H}}(f_{\mathsf{H}}(IV_0, salt \oplus \text{ipad}), \mathsf{ikm})\big),$$

where red refers to computation in GC, and green refers to local computation. Then prk is expanded to an output keying material okm with a specified length. Specifically, $\mathsf{okm} \leftarrow \mathsf{HKDF.Expand}_\ell(\mathsf{prk}, info)$ takes as input prk, a public context-specific information *info* and an output length ℓ, and outputs

$$\mathsf{okm} = T_1 \,\|\, \ldots \,\|\, T_{n-1} \,\|\, \mathsf{Trunc}_m(T_n),$$

where $T_i = \mathsf{HMAC}(\mathsf{prk}, T_{i-1} \| info \| i)$ for each $i \in [1, n]$, T_0 is an empty string, $n = \lceil \ell/256 \rceil$ and $m = \ell - 256 \cdot (n-1)$. It is easy to see that the sub-protocol Π_{PRF} (shown in the full version [65, Section B.2]) for TLS 1.2 can be directly extended to securely compute $\mathsf{HKDF.Expand}_\ell(\mathsf{prk}, info)$ for TLS 1.3. In particular, the circuit optimization for PRF in TLS 1.2 is able to be applied for $\mathsf{HKDF.Expand}$ in TLS 1.3.

In the handshake phase of TLS 1.3, the client and server run the Diffie-Hellman key exchange protocol without authentication to establish a pre-master secret pms, which can be executed similar to protocol Π_{AuthData}. Then pms is derived to a handshake secret hs via $\mathsf{HKDF.Extract}$, and then hs is derived to the client handshake traffic secret chts and server handshake traffic secret shts via $\mathsf{HKDF.Expand}$. The secrets $\mathsf{chts}, \mathsf{shts}$ are used to generate the client handshake key chk and server handshake key shk via $\mathsf{HKDF.Expand}$. Then, hs is also used to derive a mater secret ms via invoking $\mathsf{HKDF.Extract}$ and $\mathsf{HKDF.Expand}$ respective once. Next, ms is used to drive four secrets with $\mathsf{HKDF.Expand}$: the client application traffic secret cats, server application traffic secret sats, exporter master secret ems and resumption master secret rms. Using $\mathsf{HKDF.Expand}$, the secrets $\mathsf{cats}, \mathsf{sats}$ are derived to the client application key cak and server application key sak. The derivation of all secrets and keys can be securely computed by $\mathcal{P}$ and $\mathcal{V}$ by executing a protocol similar to sub-protocol Π_{PRF}.

While chk and shk are used to encrypt/decrypt the subsequent messages (e.g., client/server finished messages, signatures and certifications) in the handshake phase, cak and sak are independent and used to encrypt/decrypt application data in the record phase. Therefore, we can open chk and shk to the prover $\mathcal{P}$, and then $\mathcal{P}$ is able to *locally* perform the encryption/decryption in the handshake phase. [5] That is, it is unnecessary to run sub-protocol Π_{AEAD} (shown in the full version [65, Section B.3]) to compute stateful AEAD in the handshake phase. Besides, handshake traffic secrets chts and shts are used to produce client and server finished messages, and are independent from application traffic secrets cats and sats. In this case, we can open chts and shts to $\mathcal{P}$, and then $\mathcal{P}$ can *locally* generate the finished messages. Furthermore, ems can be computed *locally* by $\mathcal{P}$ after it knows all secrets in the post-record phase, as ems is an exporter master secret and *not* used in the record phase. The secret rms is also able to be computed *locally* by $\mathcal{P}$ after it knows all secrets in the post-record phase, if $\mathcal{P}$ and $\mathcal{V}$ will not jointly execute a session resumption with rms. The optimizations would significantly reduce the cost in the handshake phase. Overall, 21 invocations of $\mathsf{SHA256}$ compression functions need to be executed in GC-2PC, compared to that 14 invocations in TLS 1.2 for our protocol and 30 invocations in TLS 1.3 for DECO [71]. As for our protocols, the communication in the handshake phase of TLS 1.3 is about $1.5\times$ larger than that in TLS 1.2.

5 Performance Evaluation

5.1 Implementation and Experimental Setup

We implemented our protocol in C++, including 4000 lines of code of protocol development and 3000 lines of testing code. Our implementation is complete and can interact with real-world APIs. We use the EMP toolkit [61] for the implementation of the following building blocks: KOS OT [44], Ferret OT [68], half-gates-based GC with optimization of concrete security [34, 70] and interactive ZK (called QuickSilver) [66]. We leave it as the future work to incorporate the recent three-halves GC construction [56] to further reduce the communication cost of our protocol.

All benchmarks are performed over AWS m5.large instances, with 2 vCPUs and 8 GB of memory. Note that our protocol only needs about 150 MB of memory for 2KB query and response. Every experiment involves three parties: the TLS server $\mathcal{S}$, the prover $\mathcal{P}$ and the verifier $\mathcal{V}$. Except for the global-scale experiment based on real-world APIs in Section 5.3, we place $\mathcal{S}$ and $\mathcal{P}$ on the same machine and $\mathcal{V}$ on a different machine with changing network condition, where the communication between $\mathcal{S}$ and $\mathcal{P}$ is negligible compared to that between $\mathcal{P}$ and $\mathcal{V}$. We use one thread for all running time, and adopt tc to manually control the network bandwidth and roundtrip latency to desired levels. The running time and communication reported in this section are the end-to-end performance, including the preprocessing and setup costs.

[5] This observation has been found in DECO [71].

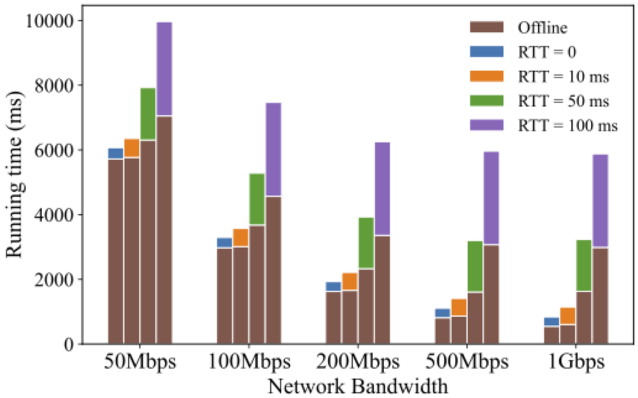

Figure 3: **Performance of our protocol under different network bandwidths and latency.** The length of query and response is 2 KB.

5.2 Scalability of Our Protocol

Performance of protocol Π_{AuthData}. In Figure 3, we show the performance of our protocol Π_{AuthData} (shown in the full version [65, Section B.4]) under different bandwidths and latency, while fixing the query and response to 2KB. We show both the offline cost (which can be done before the TLS connection) and the online cost (which can only be done during the TLS connection). Overall, our protocol is highly efficient. For example, under a realistic network with 200 Mbps bandwidth and 50 ms latency, the end-to-end running time is under four seconds while the runtime in the online phase is less than two seconds.

We can also see that the online performance is highly dependent on the latency: it is less than 50 ms when the latency is low, but could be up to 3 seconds when the latency is as high as 100 ms. This matches the roundtrip complexity that we measured from our implementation, which needs 31 roundtrips of communication. The offline cost is less affected by the latency but more on the network bandwidth; this is because the transmission of garbled circuits, which is majority of the communication of our protocol, is in the offline phase.

Comparison with prior work. We compare the performance of our protocol with DECO [71]. Since the code of DECO is not open sourced and that the performance of malicious 2PC has been constantly improving, we benchmark the performance based on the latest implementation of authenticated garbling. We also incorporate the Ferret OT [68] to the implementation to further reduce the communication cost. This is the most practically efficient malicious 2PC implementation so far. We only included the time needed in malicious 2PC, which includes computing the TLS session keys and 4 AES-GCM ciphertexts. When computing the GMAC tag, we assume that one field multiplication over $\mathbb{F}_{2^{128}}$ takes 8,765 AND gates, including 8,192 ANDs to compute the multiplication and 573 ANDs to compute the reduction. Note that there exists more efficient garbling for binary extension field multiplication [38] but only in the semi-honest setting. This is a lower bound as the DECO protocol also includes other components. All performance numbers are measured using the same type of AWS instances. The result of the comparison is shown in Table 2, where we can observe roughly 14× improvement in communication and 7.5× to 15× improvement in running time over LAN and WAN.

We record the peak memory usage of both protocols. Under 2KB query and response, the malicious 2PC needed in DECO requires a peak memory of 3 GB while our protocol only needs about 150 MB of memory. The huge difference is mainly due to the fact that authenticated garbling requires storing preprocessed triples for all AND gates in the circuit before the execution (to achieve constant roundtrips), while all building blocks that we use can be streamed without the need to store them all at once.

Performance of conversion. We also benchmarked the performance of commitment conversion of our protocol in different network settings, which is shown in Table 3. The IT-MAC-based commitments on payload is converted to Pedersen commitments [54]. We observe that in both WAN and LAN settings, the conversion protocol is very cheap compared to the overall web authentication protocol, and the cost of conversion is linear to the payload size. It takes roughly 37 ms to convert an additional kilobyte of payload to Pedersen commitments under LAN and roughly 67 ms per KB under WAN. The basetime in WAN is higher due to the higher latency.

5.3 Global-Scale Benchmarks

We integrate our protocol to access real-world web servers and test the performance, as shown in Figure 4. Specifically, we utilize provided APIs to query Coinbase and Twitter servers.

- **Coinbase API:** We benchmark fetching the balance of BTC using the prover's API secret [3]. It has a query of size 426 bytes and response of size 5701 bytes. Our protocol communicates 17.6 MB in the offline phase and 0.9 MB in the online phase.

- **Twitter API:** We benchmark using the prover's credential token to retrieve the number of followers [4]. This API has a query size of 587 bytes and response size of 894 bytes. Our protocol communicates 18.9 MB in the offline phase and 0.4 MB in the online phase.

In all experiments, the verifier $\mathcal{V}$ is deployed in the US West (represented by the purple circle), while the provers (represented by the blue circles) are distributed across 18 cities worldwide. All prover and verifier machines are hosted in AWS while the TLS server is hosted by Coinbase/Twitter, which may have nodes close to the prover. The online time required for the process ranges from 0.3 seconds to 10 seconds, depending on the round-trip time between the prover

	Communication cost				WAN (100 Mbps, RTT = 50 ms)				LAN (1 Gbps, RTT = 0 ms)			
Payload	256 B	512 B	1 KB	2 KB	256 B	512 B	1 KB	2 KB	256 B	512 B	1 KB	2 KB
DECO [71]	206 MB	255 MB	345 MB	475.7 MB	24 s	27.2 s	36.3 s	51.6 s	5.91 s	6.46 s	8.9 s	11.21 s
This work	15.2 MB	17.8 MB	22.9 MB	33.3 MB	3.19 s	3.43 s	3.96 s	4.9 s	0.46 s	0.51 s	0.61 s	0.72 s

Table 2: **Comparing the performance of DECO [71] and our protocol under LAN and WAN.**

	WAN (100 Mbps, RTT = 50 ms)				LAN (1 Gbps, RTT = 0 ms)			
Payload	256 B	512 B	1 KB	2 KB	256 B	512 B	1 KB	2 KB
Conversion	161 ms	173 ms	202 ms	278 ms	11 ms	20 ms	38 ms	76 ms

Table 3: **Performance of commitment conversion with different payload length under LAN and WAN.**

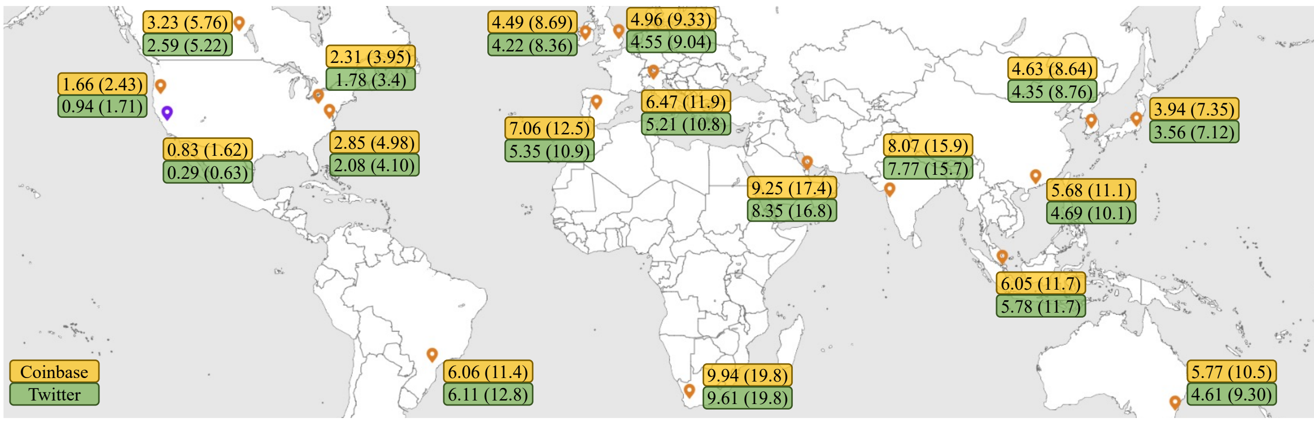

Figure 4: **Online and total performance of accessing Coinbase and Twitter servers with globally distributed provers.** All numbers are reported in seconds in the form of "online time (total time)". The verifier is fixed at California. The server is hosted by Coinbase/Twitter, which may have mirrors in various locations.

and verifier, which aligns with our expectation. From the experimental results shown in Figure 4, we conclude that our protocol is concretely efficient for real-world applications.

The performance of our protocol only depends on the bandwidth and latency in different network settings, and is independent of the concrete city in which the verifier locates. In practical scenarios, one could deploy multiple verifiers in proximity to the provers. This deployment strategy serves to minimize the round-trip time and significantly boost the overall performance of the system.

Acknowledgements

The authors would like to thank the members from TLSNotary for their helpful discussion. Kang Yang is supported by the National Natural Science Foundation of China (Grant Nos. 62102037 and 61932019). Yu Yu is supported by the National Natural Science Foundation of China (Grant Nos. 62125204 and 92270201), and the Major Program of Guangdong Basic and Applied Research (Grant No. 2019B030302008). Yu Yu's work has also been supported by the New Cornerstone Science Foundation through the XPLORER PRIZE. Xiao Wang is supported by DARPA under Contract No. HR001120C0087, NSF awards #2016240, #2236819, #2310927 and research awards from Meta and Google. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] https://www.gigamon.com/content/dam/resource-library/english/infographic/in-tls-adoption-research.pdf.

[2] https://ciphersuite.info/cs/TLS_RSA_WITH_AES_256_CBC_SHA256/.

[3] https://docs.cloud.coinbase.com/sign-in-with-coinbase/docs/api-accounts.

[4] https://developer.twitter.com/en/docs/twitter-api/users/lookup/api-reference/get-users-me.

[5] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy (S&P) 2013*, pages 526–540, 2013.

[6] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for Boolean and arithmetic circuits. In *ACM Conf. on Computer and Communications Security (CCS) 2016*, pages 565–577. ACM Press, 2016.

[7] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In *Advances in Cryptology—Crypto 2021, Part IV*, LNCS, pages 92–122. Springer, 2021.

[8] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium 2014*, pages 781–796. USENIX Association, 2014.

[9] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology—Eurocrypt 2011*, LNCS, pages 169–188. Springer, 2011.

[10] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. pages 326–349, 2012.

[11] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In *Advances in Cryptology—Crypto 2022, Part II*, LNCS, pages 603–633. Springer, 2022.

[12] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *ACM Conf. on Computer and Communications Security (CCS) 2019*, pages 291–308. ACM Press, 2019.

[13] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *Advances in Cryptology—Crypto 2019, Part III*, LNCS, pages 489–518. Springer, 2019.

[14] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In *Advances in Cryptology—Crypto 2020, Part II*, LNCS, pages 387–416. Springer, 2020.

[15] Luís T. A. N. Brandão. Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique - (extended abstract). In *Advances in Cryptology—Asiacrypt 2013, Part II*, volume 8270 of *LNCS*, pages 441–463. Springer, 2013.

[16] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High-performance multi-party computation for binary circuits based on oblivious transfer. *J. Cryptology*, 34(3):34, July 2021.

[17] Matteo Campanelli, Antonio Faonio, Dario Fiore, Anaïs Querol, and Hadrián Rodríguez. Lunar: A toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions. LNCS, pages 3–33. Springer, 2021.

[18] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In *ACM Conf. on Computer and Communications Security (CCS) 2019*, pages 2075–2092. ACM Press, 2019.

[19] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, January 2000.

[20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. LNCS, pages 738–768. Springer, 2020.

[21] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In *Advances in Cryptology—Crypto 2021, Part III*, LNCS, pages 502–534. Springer, 2021.

[22] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *Advances in Cryptology—Crypto 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, 2018.

[23] Hongrui Cui, Xiao Wang, Kang Yang, and Yu Yu. Actively secure half-gates with minimum overhead under duplex networks. LNCS, pages 35–67. Springer, 2023.

[24] Britt Cyr, Jeremy Dorfman, Ryan Hamilton, Jana Iyengar, Fedor Kouranov, Charles Krasic, Jo Kulik, Adam Langley, Jim Roskind, Robbie Shade, Satyam Shekhar, Cherie Shi, Ian Swett, Raman Tenneti, Victor Vasiliev, Antonio Vicente, Patrik Westin,

Alyssa Wilk, Dale Worley, Fan Yang, Dan Zhang, and Daniel Ziegler. QUIC wire layout specification, 2016. https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U/edit#heading=h.1jaf9kehau4e.

[25] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In *Advances in Cryptology—Crypto 2017, Part I*, volume 10401 of *LNCS*, pages 167–187. Springer, 2017.

[26] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, August 2008. http://www.rfc-editor.org/rfc/rfc5246.txt.

[27] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Authenticated garbling from simple correlations. LNCS, pages 57–87. Springer, 2022.

[28] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In *2nd Conference on Information-Theoretic Cryptography*, 2021.

[29] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In *Advances in Cryptology—Asiacrypt 2015, Part I*, LNCS, pages 711–735. Springer, 2015.

[30] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. pages 99–108. ACM Press, 2011.

[31] Niv Gilboa. Two party RSA key generation. In *Advances in Cryptology—Crypto 1999*, volume 1666 of *LNCS*, pages 116–129. Springer, 1999.

[32] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[33] Jens Groth. Short non-interactive zero-knowledge proofs. In *Advances in Cryptology—Asiacrypt 2010*, LNCS, pages 341–358. Springer, 2010.

[34] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *IEEE Symposium on Security and Privacy (S&P) 2020*, pages 825–841, 2020.

[35] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. https://www.rfc-editor.org/info/rfc6749.

[36] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In *Advances in Cryptology—Asiacrypt 2017, Part I*, LNCS, pages 598–628. Springer, 2017.

[37] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. *J. Cryptology*, 33(4):1732–1786, October 2020.

[38] David Heath and Vladimir Kolesnikov. One hot garbling. In *ACM Conf. on Computer and Communications Security (CCS) 2021*, pages 574–593. ACM Press, 2021.

[39] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021. https://www.rfc-editor.org/info/rfc9000.

[40] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In *Advances in Cryptology—Crypto 2012*, volume 7417 of *LNCS*, pages 273–293. Springer, 2012.

[41] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM Conf. on Computer and Communications Security (CCS) 2013*, pages 955–966. ACM Press, 2013.

[42] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology—Asiacrypt 2010*, LNCS, pages 177–194. Springer, 2010.

[43] Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In *Advances in Cryptology—Crypto 2018, Part III*, volume 10993 of *LNCS*, pages 365–391. Springer, 2018.

[44] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *Advances in Cryptology—Crypto 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, 2015.

[45] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *ACM Conf. on Computer and Communications Security (CCS) 2016*, pages 830–842. ACM Press, 2016.

[46] H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF). RFC 5869, 2010. https://www.rfc-editor.org/rfc/rfc5869.txt.

[47] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology—Crypto 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, 2010.

[48] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *Advances in Cryptology—Crypto 2013, Part I*, volume 8042 of *LNCS*, pages 429–448. Springer, 2013.

[49] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In *Advances in Cryptology—Crypto 2014, Part II*, volume 8617 of *LNCS*, pages 495–512. Springer, 2014.

[50] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology—Eurocrypt 2007*, LNCS, pages 52–78. Springer, 2007.

[51] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In *Advances in Cryptology—Crypto 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, 2015.

[52] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology—Crypto 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.

[53] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In *Theory of Cryptography Conference (TCC) 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, 2009.

[54] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology—Crypto 1991*, LNCS, pages 129–140. Springer, 1992.

[55] E. Rescorla. The transport layer security (TLS) protocol version 1.3. RFC 8446, August 2018. https://www.rfc-editor.org/rfc/rfc8446.txt.

[56] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In *Advances in Cryptology—Crypto 2021, Part I*, LNCS, pages 94–124. Springer, 2021.

[57] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In *Advances in Cryptology—Crypto 2022, Part I*, LNCS, pages 657–687. Springer, 2022.

[58] Nat Sakimura, John Bradley, Michael B. Jones, Breno de Medeiros, and Chuck Mortimore. OpenID Connect Core 1.0 incorporating errata set 1, 2014.

[59] abhi shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In *Advances in Cryptology—Eurocrypt 2011*, LNCS, pages 386–405. Springer, 2011.

[60] TLSNotary. Proof of data authenticity. https://docs.tlsnotary.org, Access at 2023. Source code is available at https://github.com/tlsnotary/tlsn.

[61] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[62] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *ACM Conf. on Computer and Communications Security (CCS) 2017*, pages 21–37. ACM Press, 2017.

[63] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *IEEE Symposium on Security and Privacy (S&P) 2021*, pages 1074–1091, 2021.

[64] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *USENIX Security Symposium 2021*, pages 501–518. USENIX Association, 2021.

[65] Xiang Xie, Kang Yang, Xiao Wang, and Yu Yu. Lightweight authentication of web data via garble-then-prove. Cryptology ePrint Archive, Paper 2023/964, 2023. https://eprint.iacr.org/2023/964.

[66] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *ACM Conf. on Computer and Communications Security (CCS) 2021*, pages 2986–3001. ACM Press, 2021.

[67] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In *ACM Conf. on Computer and Communications Security (CCS) 2020*, pages 1627–1646. ACM Press, 2020.

[68] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *ACM Conf. on Computer and Communications Security (CCS) 2020*, pages 1607–1626. ACM Press, 2020.

[69] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.

Functionality $\mathcal{F}_{\mathsf{OT}}$

Upon receiving $(\mathsf{ot}, (m_0, m_1))$ from a sender $\mathcal{P}$ and (ot, b) from a receiver $\mathcal{V}$, where $m_0, m_1 \in \{0,1\}^\ell$ and $b \in \{0,1\}$, this functionality outputs m_b to $\mathcal{V}$.

Figure 5: **Functionality for oblivious transfer.**

[70] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology—Eurocrypt 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, 2015.

[71] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In *ACM Conf. on Computer and Communications Security (CCS) 2020*, pages 1919–1938. ACM Press, 2020.

A More Preliminaries

OT. Oblivious Transfer (OT) allows a sender to transmit one of two messages (m_0, m_1) to a receiver, who inputs a choice bit b and obtains m_b. For security, b is kept secret against the malicious sender, and m_{1-b} is unknown for the malicious receiver. The standard OT functionality is recalled in Figure 5. Correlated OT (COT) is an important variant of OT where two messages m_0 and m_1 satisfy a fixed correlation, i.e., $m_0 \oplus m_1 = \Delta$. Both OT and COT correlations can be generated in the malicious setting using either the IKNP-like protocols [44,57] or the PCG-like protocols [12,13,68].

OLE with errors. Oblivious Linear Evaluation (OLE) can be viewed as an arithmetic generalization of OT, and allows two parties to obtain an additive sharing of multiplication of two field elements. When applying OLE into our protocol, we show that OLE with errors (OLEe) is sufficient, where the privacy is guaranteed against malicious adversaries but a malicious sender can introduce an error into the resulting OLE correlation.

Functionality for OLE with errors is shown in Figure 6. Without loss of generality, we focus on a finite field either $\mathbb{F} = \mathbb{Z}_p$ for a prime p or $\mathbb{F} = \mathbb{F}_{2^\lambda}$. We define a "gadget" vector $\boldsymbol{g} = (1, g, \ldots, g^{m-1})$ for $m = \lceil \log |\mathbb{F}| \rceil$, where $g = 2$ if $\mathbb{F} = \mathbb{Z}_p$ for a prime p and $g = X$ if $\mathbb{F} = \mathbb{F}_{2^\lambda}$. For a vector $\boldsymbol{x} \in \{0,1\}^m$, we have $\langle \boldsymbol{g}, \boldsymbol{x} \rangle = \sum_{i=1}^m x_i \cdot g^{i-1} \in \mathbb{F}$. We also denote by $\boldsymbol{g}^{-1} : \mathbb{F} \to \{0,1\}^m$ the bit-decomposition function that maps a field element $x \in \mathbb{F}$ to a bit vector $\boldsymbol{x} \in \{0,1\}^m$, such that $\langle \boldsymbol{g}, \boldsymbol{g}^{-1}(x) \rangle = x$. Following previous work (e.g., [14]), we allow a corrupted party to choose its output. If a sender $\mathcal{P}$ is corrupted, then it can introduce an error vector $\boldsymbol{e}$ into functionality $\mathcal{F}_{\mathsf{OLEe}}$. Then, $\mathcal{F}_{\mathsf{OLEe}}$ computes an error e' relying

Functionality $\mathcal{F}_{\mathsf{OLEe}}$

This functionality operates over a finite field $\mathbb{F}$. Let $m = \lceil \log |\mathbb{F}| \rceil$. This functionality interacts with a sender $\mathcal{P}$, a receiver $\mathcal{V}$ and an adversary.

- Upon receiving (ole, x) from a sender $\mathcal{P}$ and (ole, y) from a receiver $\mathcal{V}$ where $x, y \in \mathbb{F}$, execute as follows:
  1. If $\mathcal{P}$ is honest, sample $z_1 \leftarrow \mathbb{F}$. Otherwise, receive $z_1 \in \mathbb{F}$ from the adversary.
  2. If $\mathcal{P}$ is malicious, receive a vector $\boldsymbol{e} \in (\mathbb{F})^m$ from the adversary, and compute an error $e' := \langle \boldsymbol{g} * \boldsymbol{e}, \boldsymbol{y} \rangle \in \mathbb{F}$ where $\boldsymbol{y} = \boldsymbol{g}^{-1}(y)$ is the bit-decomposition of $y \in \mathbb{F}$, $*$ is a component-wise product and $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ denotes the inner product of two vectors $\boldsymbol{a}, \boldsymbol{b}$.
  3. If $\mathcal{V}$ is honest, compute $z_2 := x \cdot y - z_1 + e' \in \mathbb{F}$ (where e' is set as 0 if $\mathcal{P}$ is also honest). Otherwise, receive $z_2 \in \mathbb{F}$ from the adversary, and recompute $z_1 := x \cdot y - z_2 \in \mathbb{F}$.
- Output z_1 to $\mathcal{P}$ and z_2 to $\mathcal{V}$.

Figure 6: **Functionality for OLE with errors.**

on the input y of a receiver $\mathcal{V}$. Finally, the error e' is added into the output z_2 of $\mathcal{V}$. The introduction of errors is asymmetric, i.e., $\mathcal{V}$ is *not* allowed to add an error into the output of $\mathcal{P}$. This model the asymmetric security of the COT-based protocol [31, 45] that securely realizes functionality $\mathcal{F}_{\mathsf{OLEe}}$.

Additive secret sharings over fields. Our protocol will adopt additive secret sharings between $\mathcal{P}$ and $\mathcal{V}$ over a finite field $\mathbb{F}$. For a field element $x \in \mathbb{F}$, we write $[x] = (x_{\mathcal{P}}, x_{\mathcal{V}})$ such that $x_{\mathcal{P}} + x_{\mathcal{V}} = x \in \mathbb{F}$, where one of $x_{\mathcal{P}}, x_{\mathcal{V}}$ is random in $\mathbb{F}$. For an element $a \in \mathbb{F}$ only known by a party $\mathcal{V}$, two parties $\mathcal{V}$ and $\mathcal{P}$ can *locally* define an additive sharing $[a]_p = (a, 0)$. It is well-known that additive secret sharings are *additively homomorphic*. In particular, give public constants $c_0, c_1, \ldots, c_\ell$ and additive sharings $[x_1], \ldots, [x_\ell]$, $\mathcal{P}$ and $\mathcal{V}$ can *locally* compute $[y] := c_0 + \sum_{i=1}^{\ell} c_i \cdot [x_i]$. For an additive sharing $[x]$, we define its opening procedure:

- $x \leftarrow \mathsf{Open}([x])$: $\mathcal{P}$ sends $x_{\mathcal{P}}$ to $\mathcal{V}$, and $\mathcal{V}$ sends $x_{\mathcal{V}}$ to $\mathcal{P}$ in parallel. Then, both parties compute $x := x_{\mathcal{P}} + x_{\mathcal{V}} \in \mathbb{F}$.

For a field element x only known by $\mathcal{P}$ (resp., $\mathcal{V}$), both parties can *locally* define its additive sharing $[x] = (x, 0)$ (resp., $[x] = (0, x)$). When applying additive secret sharings into our protocol, we only need two types of finite fields: one is $\mathbb{Z}_p$ for a large prime p and the other is $\mathbb{F}_{2^{128}}$. The additive sharing of x is denoted by $[x]_p$ for former and $[x]_{2^{128}}$ for latter.