Lightweight Authentication of Web Data via Garble-Then-Prove

Xiang Xie
PADO Labs
xiexiangiscas@gmail.com

Kang Yang
State Key Laboratory of Cryptology
yangk@sklc.org

Xiao Wang
Northwestern University
wangxiao@northwestern.edu

Yu Yu
Shanghai Jiao Tong University
yuyu@cs.sjtu.edu.cn

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 Pedersen commitments so that the properties of the payload can be proven efficiently using zkSNARKs.

1 Introduction

Transport Layer Security (TLS) [DR08, Res18] 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 [ZMM$^+$20], proposed a solution that does not require any change on the TLS server side. From a high-level view, they ask the verifier, namely $\mathcal{V}$, and the prover, namely $\mathcal{P}$, to jointly emulate the computation of the TLS client $\mathcal{C}$ who interacts with $\mathcal{S}$. Since neither $\mathcal{V}$ nor $\mathcal{P}$ 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 plaintext) and desired statements on the plaintext.

Generic 2PC protocols in the malicious setting have been studied extensively in the past decade (e.g., [LP07, NO09, sS11, Bra13, LPSY15]). DECO used an implementation of the authenticated garbling [WRK17, KRRW18, YWZ20], 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 [DILO22], the computation and communication cost of maliciously secure 2PC is still much higher than its semi-honest counterparts. Moreover, these protocols often require storing preprocessed authenticated triples, thus incurring a huge memory overhead. The complexity of the 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 active 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 TLS and designing a tailored protocol for it. 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 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 $x_{\mathcal{V}}$ but it is meaningless at this point since $x_{\mathcal{V}}$ is given to $\mathcal{P}$ at this point. For obvious reasons, we call this technique *garble-then-prove*.

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

$\mathcal{P}$ region	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. $\mathcal{V}$ is always located at California.

a more than 2-fold reduction in handshake circuit size. We also pull the computation of the Galois Message Authentication Code (GMAC) out of circuits and instead use oblivious linear evaluation with errors (OLEe) [1] over $\mathbb{F}_{2^{128}}$ to compute random materials needed for GMAC, reducing the cost of GMAC computation by that a selective failure attack does not help $\mathcal{P}$ at all. more than two orders of magnitude. Doing so would allow the adversary to gain one bit of information of the TLS session secret but that would not reduce the overall concrete security, for a reason similar to prior works [KOS15, YWZ20].

Efficient commitment conversion. To prove statements on the TLS data using zkSNARKs, DECO embeds the TLS ciphertext in the statements and then proves in zkSNARKs the correct decryption. For our protocol, we use the recent vector OLE (VOLE) based zero-knowledge proofs [WYKW21, DIO21, BMRS21] during garbled-then-prove. 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 statement using VOLE-based zero-knowledge (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 [JKO13] over garbling of Boolean-to-arithmetic identity gates [BMR16]. This makes the cost conversion in the malicious case almost the same as the semi-honest setting. Then we convert arithmetic IT-MACs to Pedersen 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 Pedersen commitment efficiently, which can then be connected to various zkSNARKs [CFQ19, CFF$^+$21].

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, and can be generated more efficiently.

2 Preliminaries

We describe the TLS building blocks and the necessary cryptographic preliminaries to comprehend our protocol.

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 Appendix A.2.

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 [DR08], and is able to be extended to TLS 1.3 [Res18] that is shown in Appendix E, 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 underlying 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 $\mathsf{SHA384}$. 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 [AP13], and the GCM mode is preferred over CBC [tlsb]. 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 Appendix A, 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 pseudorandom function HMAC is defined as follows:

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

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

where opad and ipad are two public strings with length of 512 bits. 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 pseudorandom function (PRF) in TLS 1.2 [DR08], where the PRF is used to derive handshake secrets and application keys and adopts HMAC as its core. TLS 1.3 [Res18] adopts the HKDF function [Kra10, KE10] as its key derivation function, where this function is also based on HMAC. We refer the reader to Appendix E for the details of HKDF. Specifically, the PRF function with output length ℓ in TLS 1.2 is defined below:

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

where $n = \lceil \ell/256 \rceil$, $m = \ell - 256 \cdot (n-1)$, $M_1 = \mathsf{HMAC}\ (k, \mathit{label} \| \mathit{msg})$ 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 authenticated encryption with associated data (AEAD) scheme $(\mathsf{stE.Enc}, \mathsf{stE.Dec})$ to encrypt/decrypt messages in the handshake and record layers. The encryption algorithm $\mathsf{stE.Enc}(\mathsf{key}, \ell_{\textsc{c}}, \textsc{h}, \mathbf{M}, \mathsf{st}_e)$ takes as input a secret key key, a target ciphertext length $\ell_{\textsc{c}}$, a header $\textsc{h}$, a message $\mathbf{M}$ and a state st_e, and outputs a ciphertext $\textsc{ct}$. The decryption algorithm $\mathsf{stE.Dec}(\mathsf{key}, \textsc{h}, \textsc{ct}, \mathsf{st}_d)$ takes as input key, a header $\textsc{h}$, a ciphertext $\textsc{ct}$ 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 $\mathsf{AES128\_GCM}$, $\mathsf{stE.Enc}(\mathsf{key}, \ell_{\textsc{c}}, \textsc{h}, \mathbf{M}, \mathsf{st}_e)$ has the following steps:

1. Compute $\textsc{z}_0 := \mathsf{AES}(\mathsf{key}, \mathsf{st}_e)$ and update $\mathsf{st}_e := \mathsf{st}_e + 1$.
2. Suppose $\mathbf{M}$ is padded as $(\textsc{m}_1, \ldots, \textsc{m}_n)$ with $\textsc{m}_i \in \{0,1\}^{128}$. From $i = 1$ to n, compute $\textsc{z}_i := \mathsf{AES}(\mathsf{key}, \mathsf{st}_e)$ and $\textsc{c}_i := \textsc{z}_i \oplus \textsc{m}_i$ and update $\mathsf{st}_e := \mathsf{st}_e + 1$. Set $\mathbf{C} := (\textsc{c}_1, \ldots, \textsc{c}_n)$.
3. Suppose that the header $\textsc{h}$ has been padded as an element in $\mathbb{F}_{2^{128}}$. Let $\ell_{\textsc{h}}$ be the bit length of $\textsc{h}$. 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 $\sigma := \textsc{z}_0 \oplus \Phi_{(\textsc{h}, \mathbf{C}, \ell_{\textsc{h}}, \ell_{\textsc{c}})}(h)$.
4. Output $\textsc{ct} = (\mathbf{C}, \sigma)$.

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

- Parse $\textsc{ct}$ as $(\mathbf{C}, \sigma)$ and $\mathbf{C}$ as $(\textsc{c}_1, \ldots, \textsc{c}_n)$. Compute $\textsc{m}_i := \textsc{z}_i \oplus \textsc{c}_i$ for $i \in [1, n]$ and set $\mathbf{M} = (\textsc{m}_1, \ldots, \textsc{m}_n)$.
- 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 provide an overview of the standard ideal/real model [Can00, Gol04] as well as the definitions of ideal functionalities for oblivious transfer (OT), oblivious linear evaluation with errors (OLEe) and commitments in Appendix A.1.

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.**

Functionality for authenticating web data. We model the security of authenticating web data by giving an ideal functionality. In this setting, we have three roles: a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and a server $\mathcal{S}$, where $\mathcal{P}$ and $\mathcal{V}$ jointly play the role of the client to interact with the server $\mathcal{S}$. 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. 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 [ZMM$^+$20]. Following an example in [ZMM$^+$20], 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 [ZMM$^+$20].

- *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.

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 [ZMM$^+$20], we always

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

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.

2.3 Information-Theoretic MACs

Information-theoretic message authentication codes (IT-MACs) were widely used in secure multi-party computation (MPC) (e.g., [BDOZ11, NNOB12, LOS14, FKOS15, WRK17, HSS20, YWZ20, BLN+21, DILO22]) and interactive zero-knowledge (IZK) proofs [WYKW21, DIO21, BMRS21]. We will use IT-MACs to authenticate bits. Let $\Delta \in \mathbb{F}_{2^\lambda}$ be a *global* key that is only known by one party $\mathcal{V}$. A bit $x \in \{0,1\}$ known by another party $\mathcal{P}$ can be authenticated by giving $\mathcal{V}$ a uniform key $\mathsf{K}[x] \in \mathbb{F}_{2^\lambda}$ and $\mathcal{P}$ the corresponding MAC tag $\mathsf{M}[x] = \mathsf{K}[x] + x \cdot \Delta \in \mathbb{F}_{2^\lambda}$. We denote such an authenticated bit by $[\![x]\!] = (x, \mathsf{M}[x], \mathsf{K}[x])$. For a vector $\boldsymbol{x} \in \{0,1\}^n$, we write $[\![\boldsymbol{x}]\!] = (\boldsymbol{x}, \mathsf{M}[\boldsymbol{x}], \mathsf{K}[\boldsymbol{x}])$ where $\mathsf{M}[\boldsymbol{x}] = (\mathsf{M}[x_1], \ldots, \mathsf{M}[x_n])$ and $\mathsf{K}[\boldsymbol{x}] = (\mathsf{K}[x_1], \ldots, \mathsf{K}[x_n])$. IT-MACs are *additively homomorphic*, meaning that given public coefficients $c_0, c_1, \ldots, c_\ell \in \mathbb{F}_{2^\lambda}$ and IT-MACs $[\![x_1]\!], \ldots, [\![x_\ell]\!]$, $\mathcal{P}$ and $\mathcal{V}$ can *locally* compute $[\![y]\!] = \sum_{i \in [1,\ell]} c_i \cdot [\![x_i]\!] + c_0$. IT-MACs could be considered as COT correlations that can be generated by the recent PCG-like protocols [BCG+19b, BCG+19a, YWL+20, CRR21, BCG+22] with malicious security. We can also extend IT-MACs to authenticate values over a large field $\mathbb{F}$. We denote an authenticated value by $[\![x]\!]_{\mathbb{F}} = (x, \widetilde{\mathsf{M}}[x], \widetilde{\mathsf{K}}[x])$, where $\mathcal{P}$ holds $x, \widetilde{\mathsf{M}}[x] \in \mathbb{F}$ and $\mathcal{V}$ holds $\Gamma, \widetilde{\mathsf{K}}[x] \in \mathbb{F}$ such that $\widetilde{\mathsf{M}}[x] = \widetilde{\mathsf{K}}[x] + x \cdot \Gamma \in \mathbb{F}$. Authenticated bits/values can be opened and checked non-interactively in a standard way (see, e.g., [NNOB12, DNNR17, WYKW21]).

3 Technical Overview

Third-party authentication of TLS payload could be achieved using a malicious 2PC protocol with a high overhead [ZMM+20]. 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 2PC protocols based on garbled circuits that are the same as semi-honest protocols [Yao86, ZRE15, RR21] except for using malicious OT instead of semi-honest OT, and the recent interactive zero-knowledge (IZK) proofs [WYKW21, DIO21, BMRS21]. 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 [CDH+16, IT21], OAuth [Har12] and OpenID Connect [SBJ+14]. We also present a technique to convert from IT-MACs to additively homomorphic commitments that are friendly to zkSNARKs. This technique may be useful for other applications such as zero-knowledge machine learning [WYX+21]. 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.

$$
\begin{array}{lcl}
\textsc{req}_C \leftarrow \{0,1\}^{256} & \xrightarrow{\textsc{req}_C} & r_S \leftarrow \{0,1\}^{256}, t_S \leftarrow \mathbb{Z}_q \\
 & & T_S := t_S \cdot G \\
 & & \sigma_S \leftarrow \mathsf{Sign}(\mathsf{sk}_S, r_C \| r_S \| T_S) \\
\text{Verify } \mathsf{cert}_S \text{ and } \sigma_S & \xleftarrow{\textsc{res}_S} & \textsc{res}_S := (r_S, T_S, \mathsf{cert}_S, \sigma_S) \\
t_C \leftarrow \mathbb{Z}_q & & \\
T_C := t_C \cdot G & & \\
\mathsf{pms} := \mathsf{F}_x(t_C \cdot T_S) & \xrightarrow{\textsc{res}_C := T_C} & \mathsf{pms} := \mathsf{F}_x(t_S \cdot T_C)
\end{array}
$$

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

$$
\begin{array}{lcl}
\tau_C := \mathsf{H}(\textsc{req}_C, \textsc{res}_S, \textsc{res}_C) & & \\
\textsc{ufin}_C := \mathsf{PRF}(\mathsf{ms}, \mathsf{PublicStr} \| \tau_C) & & \\
\textsc{fin}_C \leftarrow \mathsf{stE.Enc}(\mathsf{key}_C, \textsc{ufin}_C, \mathsf{st}_C) & \xrightarrow{(\textsc{h}_C, \textsc{fin}_C)} & \textsc{ufin}_C \leftarrow \mathsf{stE.Dec}(\mathsf{key}_C, \textsc{fin}_C, \mathsf{st}_C) \\
 & & \text{Check } \textsc{ufin}_C \\
 & & \tau_S := \mathsf{H}(\textsc{req}_C, \textsc{res}_S, \textsc{res}_C, \textsc{ufin}_C) \\
 & & \textsc{ufin}_S := \mathsf{PRF}(\mathsf{ms}, \mathsf{PublicStr} \| \tau_S) \\
\textsc{ufin}_S \leftarrow \mathsf{stE.Dec}(\mathsf{key}_S, \textsc{fin}_S, \mathsf{st}_S) & \xleftarrow{(\textsc{h}_S, \textsc{fin}_S)} & \textsc{fin}_S \leftarrow \mathsf{stE.Enc}(\mathsf{key}_S, \textsc{ufin}_S, \mathsf{st}_S) \\
\text{Check } \textsc{ufin}_S & & \\
\textsc{enc}_Q \leftarrow \mathsf{stE.Enc}(\mathsf{key}_C, Q, \mathsf{st}_C) & \xrightarrow{(\textsc{h}_Q, \textsc{enc}_Q)} & Q \leftarrow \mathsf{stE.Dec}(\mathsf{key}_C, \textsc{enc}_Q, \mathsf{st}_C) \\
R \leftarrow \mathsf{stE.Dec}(\mathsf{key}_S, \textsc{enc}_R, \mathsf{st}_S) & \xleftarrow{(\textsc{h}_R, \textsc{enc}_R)} & \textsc{enc}_R \leftarrow \mathsf{stE.Enc}(\mathsf{key}_S, R, \mathsf{st}_S)
\end{array}
$$

Figure 2: **Graphical depiction of TLS.** PublicStr refers to strings defined in the TLS specification. Some details are omitted.

3.1 An Overview of the TLS Protocol

In Figure 2, we provide a pictorial overview, and show complete details in Figure 12 of Appendix A. 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 (pms).** $\mathcal{C}$ sends a random nonce $\textsc{req}_C \in \{0,1\}^{256}$ to $\mathcal{S}$, who samples a random nonce $r_S \in \{0,1\}^{256}$, a random element $t_S \in \mathbb{Z}_q$, and then computes a group element $T_S := t_S \cdot G$. $\mathcal{S}$ sends back $\textsc{res}_S = (r_S, t_S, \mathsf{cert}_S, \sigma_S)$, 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 $t_C \cdot T_S = t_S \cdot T_C$. The pre-master key pms is the x coordinate of this elliptic-curve group element.
- **Phase 2: TLS session keys.** With pms, $\mathcal{C}$ and $\mathcal{S}$ compute a master secret $\mathsf{ms} := \mathsf{HMAC}(\mathsf{pms}$, "master secret", $r_C \| r_S)$. Then, both parties compute a tuple $(\mathsf{key}_C, IV_C, \mathsf{key}_S, IV_S) := \mathsf{HMAC}(\mathsf{ms}$, "key expansion", $r_S \| r_C)$, where $\mathsf{key}_C, \mathsf{key}_S$ are two application keys and IV_C, IV_S would be 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.** Now, two parties exchange two rounds of test messages, which have already been known by them, over the established AEAD-encrypted channel. The client's message is $\textsc{ufin}_C = \mathsf{HMAC}(\mathsf{ms}$, "client finished", $\tau_C)$, where τ_C is the hash of the TLS transcript so far. $\mathcal{C}$ sends the AES-GCM ciphertext $\textsc{fin}_C$ of this message, which is encrypted with key_C and st_C, to $\mathcal{S}$. The server decrypts $\textsc{fin}_C$ and checks if $\textsc{ufin}_C$ is correct based on the same session keys 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 AES-GCM, except that now the underlying payload is provided by the client and the server based on the application. This phase could exchange several rounds of payload, depending on the application.

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 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 ($\textsc{req}_C$, $\textsc{res}_S$) can be done by $\mathcal{P}$ alone without $\mathcal{V}$. Since $\textsc{res}_S$ is the first message for Diffie-Hellman key exchange, the second message needs to be distributively computed by $\mathcal{P}$ and $\mathcal{V}$. In more detail, $\mathcal{P}$ and $\mathcal{V}$ pick $t_{\mathcal{V}} \leftarrow \mathbb{Z}_q$, and $t_{\mathcal{P}} \leftarrow \mathbb{Z}_q$ respectively; $\mathcal{V}$ sends $t_{\mathcal{V}} \cdot G$ to $\mathcal{P}$, who defines $\textsc{res}_C := (t_{\mathcal{P}} + t_{\mathcal{V}}) \cdot G$. The above step is similar to the previous protocols [ZMM+20, TLS23], 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 (i.e., $t_{\mathcal{V}} \cdot G$ and $t_{\mathcal{P}} \cdot G$) to an additive sharing of the x coordinate of the EC point.

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 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 abort, which could lead to c bits of loss in entropy with probability 2^{-c}. Such an attack does not reduce concrete security since the adversary could bet on c bits of the secret too. A similar analysis has been used before in designing malicious protocols (e.g., [KOS15, CDE+18, YWZ20]).

3.2.2 Phase 2: Deriving TLS session keys

Now $\mathcal{P}$ and $\mathcal{V}$ hold an additive sharing of pms and need to derive additive sharings of TLS session keys using 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× 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 garbled-circuit based 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 execution 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 this idea as an ideal functionality $\mathcal{F}_{\mathsf{GP2PC}}$ shown in Section 4.1, and show how to instantiate $\mathcal{F}_{\mathsf{GP2PC}}$ using GC-based 2PC with malicious OT and IZK, which is described in Appendix D.

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× improvement. Let's look at master secret ms as an example, which has a 384-bit output. The exact derivation formula is as follows:

$$\begin{aligned}
V &= \text{“master secret”}\|r_C\|r_S \in \{0,1\}^{592}\\
M_1 &= \mathsf{HMAC}(\mathsf{pms}, V) \in \{0,1\}^{256}\\
M_2 &= \mathsf{HMAC}(\mathsf{pms}, M_1) \in \{0,1\}^{256}\\
\mathsf{ms} &= \mathsf{HMAC}(\mathsf{pms}, M_1\|V)\|\mathsf{Trunc}_{128}\big(\mathsf{HMAC}(\mathsf{pms}, M_2\|V)\big) \in \{0,1\}^{384}
\end{aligned}$$

where $\mathsf{HMAC}(k, m) = \mathsf{SHA256}(k \oplus \text{opad}, \mathsf{SHA256}(k \oplus \text{ipad}, m))$ 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. First, $IV_1 = f_{\mathsf{H}}(IV_0, \mathsf{pms}\oplus\text{ipad})$ and $IV_2 = f_{\mathsf{H}}(IV_0, \mathsf{pms}\oplus\text{opad})$ only need to be computed once in GC 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 below, where red refers to computation in GC, green refers to local computation and blue refers to reused values.

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

where (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 takes 6 SHA256 compression calls. Later, computing $\textsc{ufin}_C$ takes another 2 compression calls in GC. As a result, the whole circuit computing all needed HMAC takes 289,827 AND gates.

3.2.3 Phase 3: Finished messages

Using a similar protocol, we compute ($\textsc{ufin}_C$, $\textsc{ufin}_S$) and reveal them to both parties. Now the main task is to perform AEAD encryption and decryption on public ciphertext/plaintext and secretly shared AEAD keys/states. 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. Note that DECO mainly supports CBC-HMAC and could support AES-GCM by computing in a Boolean circuit all c_i's and h^i's in a fully malicious 2PC. By revealing c_i to both parties while only revealing the XOR share of h^i, two parties can compute shares of the 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 $\mathbb{F}_{2^{128}}$ is 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 tag. The first task is relatively easy as we can use the above idea again to avoid malicious 2PC since the ciphertext is revealed to both parties. However, computing the tag is more complicated. Roughly speaking, the tag is an inner product between a public $\mathbb{F}_{2^{128}}$ vector and a private vector $(z_0, h^1, \ldots, h^n)$. Revealing any term in the second vector would allow the adversary to forge any message of its choice. Computing z_0 can be done in GC; however, since we reveal the XOR share 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 XOR shares of h together with z_0. Then two parties use an OLEe over $\mathbb{F}_{2^{128}}$ to compute all powers. This way, each term only needs *2KB* communication, 100× smaller than computing in GC! 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 finish messages, except that the plaintext to AES-GCM is not public anymore. Therefore, we can mostly follow the phase-3 protocol except that $\mathcal{P}$ XOR its query to the decoding information before sending to $\mathcal{V}$; this way, $\mathcal{V}$ can decode to obtain the ciphertext directly.

After $\mathcal{P}$ obtains $\textsc{enc}_R$ from the TLS server, $\mathcal{P}$ sends ($\textsc{enc}_Q$,$\textsc{enc}_R$) to $\mathcal{V}$. Then $\mathcal{V}$ sends $t_{\mathcal{V}}$ to $\mathcal{P}$, who can replay the whole TLS protocol to obtain all values computed in GC. At this point, $\mathcal{V}$ holds 1) the commitment to $\mathcal{P}$'s share of pms; 2) commitments to all values revealed from GC as XOR share; 3) the value revealed from GC 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 in GC and 2) the decryption of the ciphertext to the response. However, the cost of ZK is significantly smaller than GC: when using the latest VOLE-based ZK, the communication of ZK is only 1 bit, compared to 256 bits per gate required by GC. 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 have been verified by $\mathcal{V}$ through VOLE-based ZK. Such commitments are instantiated by IT-MACs and denoted by $[\![\boldsymbol{u}]\!]$, where Δ is the global key held by $\mathcal{V}$.

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 of $\boldsymbol{u}$, $\mathcal{V}$ 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 $\mathsf{M}[u_i] := \mathsf{H}(\mathsf{M}[u_i]) + u_i \cdot W_i$, 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 over $\mathbb{Z}_q$ (e.g., Pedersen commitment). To check consistency between IT-MACs over $\mathbb{Z}_q$ and 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 Pedersen commitments.

3.3 Protocol Summary

Previous discussions provide a high-level intuition on how we design the protocol efficiently. However, partially due to the complexity of TLS, the whole protocol is very complicated. Below, we provide a summary of the whole protocol omitting minor optimizations. The exact details of our protocol, along with the proof of security can be found in the following sections.

1. $\mathcal{P}$ samples and sends $\textsc{req}_C$ to $\mathcal{S}$ and gets back $\textsc{res}_S$.
2. $\mathcal{P}$ forwards ($\textsc{req}_C$,$\textsc{res}_S$) to $\mathcal{V}$, who sends $t_\mathcal{V} \cdot G$ to $\mathcal{P}$. $\mathcal{P}$ picks $t_\mathcal{P}$ and sends $(t_\mathcal{P} + t_\mathcal{V}) \cdot G$ to $\mathcal{S}$. Then $\mathcal{V}$ and $\mathcal{P}$ run an EC-to-Field conversion based on OLE with error so that two parties obtain additive shares of pms.
3. Two parties use Garble-Then-Prove technique so that they obtain 1) h_C, h_S and z_0's as XOR shares, 2) values of IV_C, IV_S, intermediate public value in HMAC, $\textsc{ufin}_C$, $\textsc{ufin}_S$, and the encryption of $\textsc{ufin}_C$. 3) $\mathsf{ms}, \mathsf{key}_S, \mathsf{key}_C$ in garbled format,
4. Two parties compute the tag based on OLEe over $\mathbb{F}_{2^{128}}$. $\mathcal{P}$ assembles $\textsc{fin}_C$ and sends to $\mathcal{S}$.
5. $\mathcal{P}$ forward $(\textsc{h}_S, \textsc{fin}_S)$ to $\mathcal{V}$.
6. $\mathcal{V}$ and $\mathcal{P}$ use Garble-Then-Prove to learn the ciphertext that encrypts $\mathcal{P}$'s query and XOR shares of the corresponding z_0. Two parties use OLEe to compute the tag $\mathcal{P}$ sends the the ciphertext and tag to $\mathcal{S}$; $\mathcal{S}$ returns ciphertext and tag of the response to $\mathcal{P}$, who forwards them 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 wrong, $\mathcal{P}$ 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 ZK that these commitments are consistent with the values revealed to $\mathcal{V}$ based on the TLS specification.

9. Two parties run a conversion protocol to convert VOLE-committed values to Pedersen commitments.

4 Authenticating Web Data for TLS

In Section 4.1, we define an ideal functionality $\mathcal{F}_{\mathsf{GP2PC}}$ for secure two-party computation (2PC) the garble-then-prove framework. In Appendix D, we show that combining a plain GC-based 2PC protocol with the recent IT-MACs-based IZK proof securely realizes $\mathcal{F}_{\mathsf{GP2PC}}$. [4] Based on $\mathcal{F}_{\mathsf{GP2PC}}$ we provide a complete description of our protocol that authenticates web data for TLS 1.2 in Section 4.2. This protocol 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 functionality $\mathcal{F}_{\mathsf{GP2PC}}$ 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 decrypt the ciphertext of a response R.
- **Post-record:** In this phase, the TLS protocol has terminated, and $\mathcal{S}$ does not interact with $\mathcal{P}$ and $\mathcal{V}$ any more. Now, $\mathcal{P}$ is allowed to know all the secrets and keys of the TLS session. Then $\mathcal{P}$ and $\mathcal{V}$ call $\mathcal{F}_{\mathsf{GP2PC}}$ to check the correctness of values in previous phases. Finally, both parties transform IT-MACs of Q and R into their commitments with additively homomorphic property. These commitments can then be connected to a variety of ZK proofs (e.g., zk-SNARKs).

Our main protocol Π_{AuthData} (shown in Section 4.2) invokes three sub-protocols, which are described in detail in Appendix B. Below, we show the functionalities of these sub-protocols.

- Sub-protocol Π_{E2F} (shown in Appendix 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 in the handshake phase.
- Sub-protocol Π_{PRF} (shown in Appendix B.2) calls $\mathcal{F}_{\mathsf{GP2PC}}$ to compute HMAC-based PRF in the handshake phase. Then, it proves correctness of all values opened 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 $\textsc{ufin}_C, \textsc{ufin}_S$.
- Sub-protocol Π_{AEAD} (shown in Appendix B.3) calls $\mathcal{F}_{\mathsf{GP2PC}}$ to compute AES blocks used for encryption and decryption of stateful 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. Π_{AEAD} is used to encrypt $\textsc{ufin}_C, Q$ to obtain the ciphertexts $\textsc{fin}_C, \textsc{enc}_Q$ and decrypt $\textsc{fin}_S$ to get $\textsc{ufin}_S$.

$\mathcal{P}$ and $\mathcal{V}$ generate authenticated bits $[\![Q]\!]$ and $[\![R]\!]$ in the post-record phase by calling an ideal functionality $\mathcal{F}_{\mathsf{IZK}}$ for IZK proofs based on IT-MACs. Functionality $\mathcal{F}_{\mathsf{IZK}}$ is a simple extension of the ideal functionality defined in [WYX$^+$21]. In Appendix A.3, we give the details of $\mathcal{F}_{\mathsf{IZK}}$ and show that the recent IZK protocols [WYKW21, DIO21, BMRS21] based on IT-MACs securely realize $\mathcal{F}_{\mathsf{IZK}}$. We assume that $\mathcal{F}_{\mathsf{IZK}}$ also inherits the commands of $\mathcal{F}_{\mathsf{GP2PC}}$, such that it can directly authenticate the inputs and outputs stored in $\mathcal{F}_{\mathsf{GP2PC}}$ with IT-MACs, where this works if both $\mathcal{F}_{\mathsf{GP2PC}}$ and $\mathcal{F}_{\mathsf{IZK}}$ adopt the IT-MACs-based IZK protocol to instantiate. Besides, $\mathcal{P}$ and $\mathcal{V}$ call an ideal functionality $\mathcal{F}_{\mathsf{Conv}}$ (shown in Appendix C) to convert $[\![Q]\!]$ and $[\![R]\!]$ into their additively

[4] Recall that a plain GC-based 2PC protocol is the same as the semi-honest protocol [Yao86, ZRE15, RR21], except for using malicious OT rather than semi-honest OT.

Functionality $\mathcal{F}_{\mathsf{GP2PC}}$

This functionality runs with a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and an adversary, and initializes a state $\mathsf{state}_0 = \bot$.

Input. Upon receiving $(\mathsf{input}, \mathsf{id}, x)$ from $A \in \{\mathcal{P}, \mathcal{V}\}$ and $(\mathsf{input}, \mathsf{id})$ from the other party, where id is a fresh identifier and $x \in \{0,1\}$, store (id, x).

Evaluate. Upon receiving $(\mathsf{eval}, j, f_j, \mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3)$ from $\mathcal{P}$ and $\mathcal{V}$, where this is the j-th call and $f_j : \{0,1\}^* \times \{0,1\}^m \times \{0,1\}^m \to \{0,1\}^* \times \{0,1\}^n$ is a Boolean circuit, do the following:

1. If $\mathsf{id}_{1,i}$ (resp., $\mathsf{id}_{2,i}$) for all $i \in [1,m]$ are present in memory, retrieve $(\mathsf{id}_{1,i}, x_{j,i})$ (resp., $(\mathsf{id}_{2,i}, y_{j,i})$) for $i \in [1,m]$, and define $\boldsymbol{x}_j = (x_{j,1}, \ldots, x_{j,m})$ (resp., $\boldsymbol{y}_j = (y_{j,1}, \ldots, y_{j,m})$). If $\mathsf{id}_{1,i} = \bot$ (resp., $\mathsf{id}_{2,i} = \bot$) for all $i \in [1,m]$, set $\boldsymbol{x}_j = \bot$ (resp., $\boldsymbol{y}_j = \bot$). Otherwise, abort.
2. If $\mathcal{P}$ is honest, set $(\mathsf{state}_j, \boldsymbol{z}_j) := f_j(\mathsf{state}_{j-1}, \boldsymbol{x}_j, \boldsymbol{y}_j)$. Otherwise, receive a circuit f'_j from the adversary, and compute $(\mathsf{state}_j, \boldsymbol{z}_j) := f'_j(\mathsf{state}_{j-1}, \boldsymbol{y}_j)$.
3. Update the state as state_j and store $(\mathsf{id}_{3,i}, z_{j,i})$ for $i \in [1,n]$.

Output. Upon receiving $(\mathsf{output}, \mathsf{id}, A)$ from both parties where $A \in \{\mathcal{P}, \mathcal{V}\}$ where (id, z) was previously stored, update (id, z) as $(\mathsf{id}, z, \mathit{output}, A)$. If $\mathcal{V}$ is corrupted and $A = \mathcal{P}$, receive $e \in \{0,1\}$ from the adversary and output $(\mathsf{id}, z \oplus e)$ to $\mathcal{P}$. Otherwise, output $(\mathsf{output}, \mathsf{id}, z)$ to A.

Reveal and Prove. Upon receiving $(\mathsf{revealandprove})$ from $\mathcal{P}$ and $\mathcal{V}$, send all $\mathcal{V}$'s inputs to $\mathcal{P}$, and ignore any (eval) command.Then, from $j = 1$ to ℓ where ℓ is the number of calls to the eval command, execute as follows:

1. Receive $(\mathsf{prove}, j, g_j, \mathbf{id}_j)$ from $\mathcal{P}$ and $\mathcal{V}$, where g_j is the verification circuit corresponding to the circuit f_j and $\mathbf{id}_j$ is the vector of identifiers on output $\boldsymbol{z}_j$ in the j-th (eval) call.
2. Set $\boldsymbol{x}_j$ as the input vector in the j-th (eval) call. Run $(\mathsf{state}^*_j, \boldsymbol{z}^*_j) := g_j(\mathsf{state}^*_{j-1}, \boldsymbol{x}_j)$ where $\mathsf{state}^*_0 = \bot$.
3. For each $i \in [1,n]$, if $(\mathsf{id}_{j,i}, z_{j,i}, \mathit{output}, \mathcal{V})$ was previously stored, then check that $z^*_{j,i} = z_{j,i}$. If any check fails, send $(\mathsf{prove}, j, \mathsf{false})$ to $\mathcal{V}$. Otherwise, send $(\mathsf{prove}, j, \mathsf{true})$ to $\mathcal{V}$.

Figure 3: **Reactive functionality for 2PC in the garble-then-prove framework.**

homomorphic commitments in the post-record phase. In Appendix C, we present an efficient protocol to securely realize $\mathcal{F}_{\mathsf{Conv}}$. In Appendix E, we show how to extend our protocol to support TLS 1.3.

4.1 Functionality for Garble-Then-Prove

In Figure 3, we give the detailed definition of ideal functionality $\mathcal{F}_{\mathsf{GP2PC}}$ in the garble-then-prove framework. In particular, the first three commands model the security of garbled-circuits-based secure two-party computation (GC-2PC) in the semi-honest setting and the last command abstracts the security of ZK proofs. In the real protocol execution, $\mathcal{P}$ plays the role of the garbler and prover, while $\mathcal{V}$ acts as the evaluator and verifier. After a GC-2PC protocol execution, we always consider that $\mathcal{P}$ could obtain all inputs (including $\mathcal{V}$'s inputs) and secrets. Then, $\mathcal{P}$ can convince $\mathcal{V}$ that all the values obtained by $\mathcal{V}$ are correct using a ZK proof. This makes our garble-then-prove approach (defined in $\mathcal{F}_{\mathsf{GP2PC}}$) not suitable for the case that the inputs of $\mathcal{V}$ need to be kept secret in the whole protocol execution. In the next subsection, we will show that $\mathcal{F}_{\mathsf{GP2PC}}$ is able to be used to authenticate web data for TLS. More applications of our garble-then-prove approach can be exploited, e.g., authenticating data for QUIC [CDH+16, IT21], OAuth [Har12] and OpenID Connect [SBJ+14].

By calling the (input) command, one of two parties $\mathcal{P}$ and $\mathcal{V}$ is able to input a bit. $\mathcal{F}_{\mathsf{GP2PC}}$ defines a reactive functionality, which allows two parties to evaluate a series of Boolean circuits $f_1, \ldots, f_\ell$, such that the input to each circuit f_j is the state information state_{j-1}, $\mathcal{P}$'s input vector

Protocol Π_{AuthData}

A prover $\mathcal{P}$ and a verifier $\mathcal{V}$, who execute the following protocol to play the role of a client, interact with a server $\mathcal{S}$ who inputs $(\mathsf{cert}_S, \mathsf{sk}_S)$ defined in Figure 12. $\mathcal{P}$ holds a private input α for a query template Query known by both parties.

- **Preprocessing : Generate correlated randomness.** Before starting a TLS session, $\mathcal{P}$ and $\mathcal{V}$ run the preprocessing phase of sub-protocols Π_{E2F} (shown in Figure 13) and Π_{AEAD} (shown in Figure 15) to generate additive sharings on random values and their computational results. Both parties call functionality $\mathcal{F}_{\mathsf{IZK}}$ to initialize a global key $\Delta \in \{0,1\}^{\lambda}$ sampled uniformly by $\mathcal{V}$. (When $\mathcal{F}_{\mathsf{GP2PC}}$ and $\mathcal{F}_{\mathsf{IZK}}$ are instantiated, garbled circuits and random IT-MACs are generated in this phase.)
- **Handshake : Generate pre-master secret.**
  1. $\mathcal{P}$ samples $r_C \leftarrow \{0,1\}^{256}$ and sends $\textsc{req}_C := r_C$ to $\mathcal{S}$. Then, $\mathcal{P}$ receives $\textsc{res}_S = (r_S, T_S, \mathsf{cert}_S, \sigma_S)$ from $\mathcal{S}$, and sends $(\textsc{req}_C, \textsc{res}_S)$ to $\mathcal{V}$. If cert_S is invalid or $\mathsf{Verify}(\mathsf{pk}_S, r_C \| r_S \| T_S, \sigma_S) = 0$, $\mathcal{P}$ and $\mathcal{V}$ abort.
  2. $\mathcal{V}$ samples $t_{\mathcal{V}} \leftarrow \mathbb{Z}_q$, computes $T_{\mathcal{V}} := t_{\mathcal{V}} \cdot G$, and sends $T_{\mathcal{V}}$ to $\mathcal{P}$. $\mathcal{P}$ samples $t_{\mathcal{P}} \leftarrow \mathbb{Z}_q$ and computes $T_{\mathcal{P}} := t_{\mathcal{P}} \cdot G$, and sets $T_C := T_{\mathcal{P}} + T_{\mathcal{V}} = (t_{\mathcal{P}} + t_{\mathcal{V}}) \cdot G$. Then, $\mathcal{P}$ sends $\textsc{res}_C := T_C$ to $\mathcal{S}$ and $\mathcal{V}$.
  3. $\mathcal{P}$ computes $Z_1 := t_{\mathcal{P}} \cdot T_S$, and $\mathcal{V}$ computes $Z_2 := t_{\mathcal{V}} \cdot T_S$. Then $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol Π_{E2F} (shown in Figure 13) on respective input Z_1 and Z_2 to get an additive sharing $[\widetilde{\mathsf{pms}}]_p = \mathsf{F}_x(Z_1 + Z_2)$, where $\mathcal{P}$ holds $\widetilde{\mathsf{pms}}_{\mathcal{P}} \in \mathbb{Z}_p$ and $\mathcal{V}$ has $\widetilde{\mathsf{pms}}_{\mathcal{V}} \in \mathbb{Z}_p$ such that $\widetilde{\mathsf{pms}}_{\mathcal{P}} + \widetilde{\mathsf{pms}}_{\mathcal{V}} = \widetilde{\mathsf{pms}} \mod p$.
- **Handshake : Generate master secret and application keys.**
  4. $\mathcal{P}$ and $\mathcal{V}$ define the bit decomposition of $\widetilde{\mathsf{pms}}_{\mathcal{P}} \in \mathbb{Z}_p$ and $\widetilde{\mathsf{pms}}_{\mathcal{V}} \in \mathbb{Z}_p$ as $\mathsf{pms}_{\mathcal{P}} \in \{0,1\}^{\lceil \log p \rceil}$ and $\mathsf{pms}_{\mathcal{V}} \in \{0,1\}^{\lceil \log p \rceil}$ respectively. Let *AddModp* be a Boolean circuit which inputs $\mathsf{pms}_{\mathcal{P}}, \mathsf{pms}_{\mathcal{V}} \in \{0,1\}^{\lceil \log p \rceil}$, and outputs $\mathsf{pms} \in \{0,1\}^{\lceil \log p \rceil}$ that is the bit decomposition of $\widetilde{\mathsf{pms}} = \widetilde{\mathsf{pms}}_{\mathcal{P}} + \widetilde{\mathsf{pms}}_{\mathcal{V}} \in \mathbb{Z}_p$. Then, both parties call the (eval) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on $(\mathsf{pms}_{\mathcal{P}}, \mathsf{pms}_{\mathcal{V}})$ and *AddModp* to generate and store pms, where $\mathsf{pms}_{\mathcal{P}}$ is committed by the (input) command of $\mathcal{F}_{\mathsf{GP2PC}}$.
  5. Both parties run sub-protocol $\Pi^{(1)}_{\mathsf{PRF}}$ (shown in Figure 14) on pms, ("master secret", $r_C \| r_S$) and a label "secret" to generate $\mathsf{ms} = \mathsf{PRF}_{384}(\mathsf{pms}, \text{"master secret"}, r_C \| r_S)$, where $\mathcal{F}_{\mathsf{GP2PC}}$ stores ms. Note that no one knows pms and ms in this phase.
  6. $\mathcal{P}$ and $\mathcal{V}$ execute sub-protocol $\Pi^{(2)}_{\mathsf{PRF}}$ (shown in Figure 14) on ms, ("key expansion", $r_S \| r_C$) and a label "partial open" to generate 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 $\mathcal{F}_{\mathsf{GP2PC}}$ stores $\mathsf{key}_C, \mathsf{key}_S \in \{0,1\}^{128}$ and opens $IV_C, IV_S \in \{0,1\}^{96}$ to both parties. Both parties initialize $(\mathsf{st}^C_e, \mathsf{st}^C_d) := (IV_C, IV_S)$.
- **Handshake : Exchange finished messages.**
  7. $\mathcal{P}$ and $\mathcal{V}$ compute $\tau_C := \mathsf{H}(\textsc{req}_C \| \textsc{res}_S \| \textsc{res}_C)$. Then, both parties run sub-protocol $\Pi^{(3)}_{\mathsf{PRF}}$ (shown in Figure 14) on ms, a pair ("client finished", τ_C) and a label "open" to let the parties obtain $\textsc{ufin}_C = \mathsf{PRF}_{96}(\mathsf{ms}, \text{"client finished"}, \tau_C)$.
  8. $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi^{(1)}_{\mathsf{AEAD}}$ (shown in Figure 15) on key_C, a tuple $(\mathsf{st}^C_e, \ell_C, \textsc{h}_C, \textsc{ufin}_C)$, a label "encryption" and a label "open" to generate $\textsc{fin}_C = \mathsf{stE.Enc}(\mathsf{key}_C, \ell_C, \textsc{h}_C, \textsc{ufin}_C, \mathsf{st}^C_e)$. During this execution, both parties obtain $[h^i_C]_{2^{128}}$ for all $i \in [1, m]$ where $h_C = \mathsf{AES}(\mathsf{key}_C, \mathbf{0})$ and $m = \lceil |Q|/128 \rceil + 2$. Both parties update $\mathsf{st}^C_e := \mathsf{st}^C_e + 2$. Then, $\mathcal{P}$ sends $(\textsc{h}_C, \textsc{fin}_C)$ to $\mathcal{S}$ who checks the correctness of $\textsc{fin}_C$ and $\textsc{ufin}_C$ following the TLS specification and aborts if the check fails.
  9. $\mathcal{P}$ and $\mathcal{V}$ compute $\tau_S := \mathsf{H}(\textsc{req}_C \| \textsc{res}_S \| \textsc{res}_C \| \textsc{ufin}_C)$. Then, both parties run sub-protocol $\Pi^{(4)}_{\mathsf{PRF}}$ (shown in Figure 14) on ms, a pair ("server finished", τ_S) and label "open" such that they get $\textsc{ufin}_S = \mathsf{PRF}_{96}(\mathsf{ms}, \text{"server finished"}, \tau_S)$.
  10. After receiving $(\textsc{h}_S, \textsc{fin}_S)$ from $\mathcal{S}$, $\mathcal{P}$ forwards it to $\mathcal{V}$. Then, $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi^{(2)}_{\mathsf{AEAD}}$ (shown in Figure 15) on key_S, a tuple $\left(\mathsf{st}^C_d, \ell_S, \textsc{h}_S, \textsc{fin}_S\right)$, a label "decryption" and a label "open" such that both parties get $\textsc{ufin}_S{}'$. Then, $\mathcal{P}$ and $\mathcal{V}$ check that $\textsc{ufin}_S = \textsc{ufin}_S{}'$ and abort if the check fails. The parties update $\mathsf{st}^C_d := \mathsf{st}^C_d + 2$.

Figure 4: **Protocol of authenticating web data for TLS.**

Protocol Π_{AuthData}, continued

- **Record : Query and Respond.** $\mathcal{P}$ and $\mathcal{V}$ encrypt a query and obtain a ciphertext on a response.

11. $\mathcal{P}$ computes $Q := \mathsf{Query}(\alpha)$. Then, $\mathcal{P}$ and $\mathcal{V}$ execute sub-protocol $\Pi^{(3)}_{\mathsf{AEAD}}$ (shown in Figure 15) on key_C, a tuple $\big(\mathsf{st}^C_e, \ell_Q, \textsc{h}_Q, Q, \{[h^i_C]_{2^{128}}\}_{i\in[1,m]}\big)$, a label "encryption" and a label "secret" such that both parties obtain $\textsc{enc}_Q = \mathsf{stE.Enc}(\mathsf{key}_C, \ell_Q, \textsc{h}_Q, Q, \mathsf{st}^C_e)$. Then, $\mathcal{P}$ sends $(\textsc{h}_Q, \textsc{enc}_Q)$ to $\mathcal{S}$.
12. Following the TLS specification, $\mathcal{S}$ checks correctness of $(\textsc{h}_Q, \textsc{enc}_Q)$, and aborts if the check fails. Then $\mathcal{S}$ decrypts $\textsc{enc}_Q$ to get Q, and computes a ciphertext $(\textsc{h}_R, \textsc{enc}_R)$ on a response R. Then, $\mathcal{S}$ sends $(\textsc{h}_R, \textsc{enc}_R)$ to $\mathcal{P}$, who forwards it to $\mathcal{V}$.

- **Post-record : Prove with ZK.** $\mathcal{P}$ and $\mathcal{V}$ check correctness of the values produced so far.

13. Both parties run the post-record phase of sub-protocol executions $\Pi^{(1)}_{\mathsf{AEAD}}$, $\Pi^{(2)}_{\mathsf{AEAD}}$ and $\Pi^{(3)}_{\mathsf{AEAD}}$ to let $\mathcal{V}$ obtain $(h_C, h_S, \textsc{z}_C, \textsc{z}_S, \textsc{z}_Q)$, where $\textsc{z}_C, \textsc{z}_S, \textsc{z}_Q$ are the AES blocks used to generate the GMAC tags involved in $\textsc{fin}_C, \textsc{fin}_S, \textsc{enc}_Q$.
14. $\mathcal{P}$ and $\mathcal{V}$ call the $(\mathsf{revealandprove})$ command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$, and $\mathcal{P}$ obtains $\mathsf{pms}_\mathcal{V}$. In parallel, $\mathcal{V}$ sends $t_\mathcal{V} \in \mathbb{Z}_p$ to $\mathcal{P}$. Then $\mathcal{P}$ verifies that $T_\mathcal{V} = t_\mathcal{V} \cdot G$ and aborts if the equality does not hold. $\mathcal{P}$ performs the following checks, and sends abort to $\mathcal{F}_{\mathsf{GP2PC}}$ and aborts if any check fails.
    (a) $\mathcal{P}$ computes $\widetilde{\mathsf{pms}}^* := \mathsf{F}_x((t_\mathcal{P} + t_\mathcal{V}) \cdot T_S)$, sets pms^* as the bit decomposition of $\widetilde{\mathsf{pms}}^*$, and then checks that $\mathsf{pms}^* = AddModp(\mathsf{pms}_\mathcal{P}, \mathsf{pms}_\mathcal{V})$.
    (b) $\mathcal{P}$ uses pms^* to check correctness of all values opened during $\Pi^{(1)}_{\mathsf{PRF}}$, by recomputing these values with pms^* and comparing them with the values opened.
    (c) $\mathcal{P}$ computes $\mathsf{ms}^* := \mathsf{PRF}_{384}(\mathsf{pms}^*, \text{“master secret”}, r_C \| r_S)$, and uses ms^* to check the correctness of all values opened during sub-protocol executions $\Pi^{(2)}_{\mathsf{PRF}}$, $\Pi^{(3)}_{\mathsf{PRF}}$ and $\Pi^{(4)}_{\mathsf{PRF}}$ by recomputing these values with ms^*, where $(\mathsf{key}^*_C, IV^*_C, \mathsf{key}^*_S, IV^*_S) := \mathsf{PRF}_{320}(\mathsf{ms}^*, \text{“key expansion”}, r_S \| r_C)$ is computed, $IV_C = IV^*_C, IV_S = IV^*_S$ are checked, and $\textsc{ufin}_C, \textsc{ufin}_S$ are checked.
    (d) $\mathcal{P}$ uses $(\mathsf{key}^*_C, \mathsf{key}^*_S, \textsc{ufin}_C, \textsc{ufin}_S, Q)$ to check correctness of $\textsc{fin}_C$, $\textsc{fin}_S$ and $\textsc{enc}_Q$ by recomputing these ciphertexts and comparing them with the ciphertexts $\textsc{fin}_C$, $\textsc{fin}_S$ and $\textsc{enc}_Q$.
    (e) $\mathcal{P}$ runs $R \leftarrow \mathsf{stE.Dec}(\mathsf{key}_S, \textsc{h}_R, \textsc{enc}_R, \mathsf{st}^C_d)$. If $\mathsf{stE.Dec}$ outputs $\perp$, $\mathcal{P}$ aborts.
15. Let $\overline{AddModp}$ be a Boolean circuit which inputs $\mathsf{pms}_\mathcal{P}$ and outputs $\mathsf{pms}^* = AddModp(\mathsf{pms}_\mathcal{P}, \mathsf{pms}_\mathcal{V})$ for a common value $\mathsf{pms}_\mathcal{V}$. $\mathcal{P}$ and $\mathcal{V}$ call the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on $\mathsf{pms}_\mathcal{P}$ and circuit $\overline{AddModp}$ to generate and store pms^*, where $\mathsf{pms}_\mathcal{P}$ has been committed in the handshake phase.
16. Given pms^* stored in $\mathcal{F}_{\mathsf{GP2PC}}$, by calling $\mathcal{F}_{\mathsf{GP2PC}}$, $\mathcal{P}$ and $\mathcal{V}$ execute the post-record phase of sub-protocol executions $\Pi^{(1)}_{\mathsf{PRF}}$, $\Pi^{(2)}_{\mathsf{PRF}}$, $\Pi^{(3)}_{\mathsf{PRF}}$, $\Pi^{(4)}_{\mathsf{PRF}}$, $\Pi^{(1)}_{\mathsf{AEAD}}$, $\Pi^{(2)}_{\mathsf{AEAD}}$ and $\Pi^{(3)}_{\mathsf{AEAD}}$ to let $\mathcal{V}$ check the correctness of all values obtained by $\mathcal{V}$ in the previous steps.
17. Both parties call the (zkauth) command of $\mathcal{F}_{\mathsf{IZK}}$ on key_S and Boolean circuit $AES[\mathsf{st}^C_d]$ to generate $\llbracket \textsc{z}_R \rrbracket$, where $AES[\mathsf{st}^C_d]$ inputs key_S and outputs $\textsc{z}_R = \mathsf{AES}(\mathsf{key}_S, \mathsf{st}^C_d)$. $\mathcal{P}$ sends $\textsc{z}_R$ to $\mathcal{V}$, and then both parties call the (check) command of $\mathcal{F}_{\mathsf{IZK}}$ on input $\llbracket \textsc{z}_R \rrbracket - \textsc{z}_R$ to verify that $\textsc{z}_R$ received by $\mathcal{V}$ is correct. Then, $\mathcal{V}$ uses $(h_C, h_S, \textsc{z}_C, \textsc{z}_S, \textsc{z}_Q, \textsc{z}_R)$ to check the correctness of all GMAC tags in AEAD ciphertexts $\textsc{fin}_C, \textsc{fin}_S, \textsc{enc}_Q, \textsc{enc}_R$ following the AEAD specification, and aborts if the check fails.
18. Let $n = \lceil |Q|/128 \rceil$. For $i \in [1, n]$, $\mathcal{P}$ and $\mathcal{V}$ call the $(\mathsf{authinput})$ command of $\mathcal{F}_{\mathsf{IZK}}$ on $\textsc{z}_{i,\mathcal{P}}$ to generate $\llbracket \textsc{z}_{i,\mathcal{P}} \rrbracket$, where $\mathcal{P}$ and $\mathcal{V}$ hold $\textsc{z}_{i,\mathcal{P}}$ and $\textsc{z}_{i,\mathcal{V}}$ such that $\textsc{z}_{i,\mathcal{P}} \oplus \textsc{z}_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}_C, \mathsf{st}^C_e + i)$, and $\textsc{z}_{i,\mathcal{P}}$ has been committed in the preprocessing phase. Both parties locally compute $\llbracket \textsc{z}_i \rrbracket := \llbracket \textsc{z}_{i,\mathcal{P}} \rrbracket \oplus \textsc{z}_{i,\mathcal{V}}$ for $i \in [1, n]$. Then, $\mathcal{P}$ and $\mathcal{V}$ parse $\textsc{enc}_Q = (\mathbf{C}_Q, \sigma_Q)$ and $\mathbf{C}_Q = (\textsc{c}_1, \ldots, \textsc{c}_n)$. Both parties locally compute $\llbracket \textsc{m}_i \rrbracket := \llbracket \textsc{z}_i \rrbracket \oplus \textsc{c}_i$ for $i \in [1, n]$, and set $\llbracket Q \rrbracket = (\llbracket \textsc{m}_1 \rrbracket, \ldots, \llbracket \textsc{m}_n \rrbracket)$.
19. $\mathcal{P}$ and $\mathcal{V}$ parse $\textsc{enc}_R = (\mathbf{C}_R, \sigma_R)$. Both parties call the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on key_S and circuit $AESDec[\mathsf{st}^C_d, \mathbf{C}_R]$ to generate $\llbracket R \rrbracket$, where $AESDec[\mathsf{st}^C_d, \mathbf{C}_R]$ takes as input key_S, and decrypts $\mathbf{C}_R$ to an output R with key_S and st^C_d.

- **Post-record : Commit to query and response.**

20. $\mathcal{P}$ and $\mathcal{V}$ call the $(\mathsf{convert})$ command of functionality $\mathcal{F}_{\mathsf{Conv}}$ (shown in Figure 17) on $(\llbracket Q \rrbracket, \llbracket R \rrbracket)$ to obtain $(\mathsf{cid}_1, \ldots, \mathsf{cid}_\ell)$ (resp., $(\mathsf{cid}'_1, \ldots, \mathsf{cid}'_n)$) that denotes the commitment identifiers on Q (resp., R). Then, both parties output these identifiers, and $\mathcal{P}$ also outputs (Q, R).

Figure 5: **Protocol of authenticating web data for TLS, continued.**

$\boldsymbol{x}_j$ and $\mathcal{V}$'s input vector $\boldsymbol{y}_j$, and the output includes the updated state information state_j and output vector $\boldsymbol{z}_j$. If $\mathcal{P}$ is corrupted, $\mathcal{F}_{\mathsf{GP2PC}}$ receives an arbitrary circuit f'_j from the adversary, and computes the output with f'_j. In this case, since f'_j is totally determined by the adversary, it can have involved $\mathcal{P}$'s input, and thus the input to circuit f'_j does not contain $\boldsymbol{x}_j$. Through the (output) command, $\mathcal{P}$ or $\mathcal{V}$ can obtain a circuit output. If both parties use the (output) command with the same identifier to get an output, this is equivalent to open the output to both parties. If $\mathcal{V}$ is corrupted, we allow the adversary to add an error into the bit output to $\mathcal{P}$. By calling the ($\mathsf{revealandprove}$) command, $\mathcal{P}$ gets all inputs (including $\mathcal{V}$'s inputs) from $\mathcal{F}_{\mathsf{GP2PC}}$. In this case, $\mathcal{P}$ is able to check the correctness of all the values outputted to it by recomputing these values. If the check fails, $\mathcal{P}$ could send abort to functionality $\mathcal{F}_{\mathsf{GP2PC}}$. This command allows $\mathcal{P}$ to convince $\mathcal{V}$ that all values output to $\mathcal{V}$ are correct. In particular, $\mathcal{F}_{\mathsf{GP2PC}}$ uses a verification circuit g_j to check the output computed by circuit f'_j if $\mathcal{P}$ is corrupted or f_j otherwise. If $\mathcal{P}$ is honest, the check always passes. If $\mathcal{P}$ is corrupted and provides an incorrect circuit f'_j leading to a different output, then the check fails and $\mathcal{F}_{\mathsf{GP2PC}}$ would abort. We always consider that the input $\boldsymbol{x}_j$ for the j-th (eval) call is used as the input of circuit g_j. This holds even if $\boldsymbol{x}_j$ is *not* used for f'_j chosen by a malicious $\mathcal{P}$. In this case, we can view $\boldsymbol{x}_j$ as being committed by the (input) command. We can define $g_j(\mathsf{state}^*_{j-1}, \boldsymbol{x}_j) = f_j(\mathsf{state}^*_{j-1}, \boldsymbol{x}_j, \boldsymbol{y}_j)$ where $\mathcal{V}$'s input $\boldsymbol{y}_j$ is known by both parties in the reveal-and-prove phase and has been involved in g_j, and $\mathsf{state}^*_{j-1} = \mathsf{state}_{j-1}$ in the honest case. If $\boldsymbol{x}_j$ is public, then $\boldsymbol{x}_j$ can be defined in g_j and $g_j(\mathsf{state}^*_{j-1}, \boldsymbol{x}_j)$ ignores the input $\boldsymbol{x}_j$ in this case. We always assume that any circuit output $\boldsymbol{z}_j$ is not input to the next circuit f_{j+1}, as $\boldsymbol{z}_j$ can be included in state_j and then state_j is able to be used by f_{j+1}.

Before the ($\mathsf{revealandprove}$) command is called, the bits output to $\mathcal{P}$ or $\mathcal{V}$ may be incorrect. After the ($\mathsf{revealandprove}$) command was executed, $\mathcal{P}$ checks the correctness of all its outputs by itself, and the correctness of the circuit outputs obtained by $\mathcal{V}$ is checked. That is, functionality $\mathcal{F}_{\mathsf{GP2PC}}$ can guarantee the correctness of circuit evaluations at the end. Besides, $\mathcal{F}_{\mathsf{GP2PC}}$ assures the privacy of the honest party's inputs, as the messages between $\mathcal{F}_{\mathsf{GP2PC}}$ and the honest party are communicated over a secure channel. However, if $\mathcal{P}$ is corrupted, it is able to mount a selective-failure attack. Informally, given a circuit f and inputs x, y of $\mathcal{P}$ and $\mathcal{V}$, a malicious $\mathcal{P}$ could learn $f'(x', y) = f(x, y)$ for the malicious chosen circuit f' and input x'. This leaks at most one-bit information on the input y of honest party $\mathcal{V}$. Note that this is harmless as $\mathcal{P}$ could always get y from the ($\mathsf{revealandprove}$) command. If $\mathcal{P}$ is honest, a malicious $\mathcal{V}$ cannot learn any secret information.

4.2 Main Protocol

In Figures 4 and 5, we present the details of main protocol Π_{AuthData}, which enables $\mathcal{P}$ to convince $\mathcal{V}$ that a query Q and a response R committed are consistent, i.e., R is produced by a TLS server on Q. The main protocol invokes three sub-protocols: Π_{E2F}, Π_{PRF}, Π_{AEAD}, and works in the $(\mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{IZK}}, \mathcal{F}_{\mathsf{Conv}})$-hybrid model, where $\mathcal{F}_{\mathsf{OLEe}}$ is called by $\Pi_{\mathsf{E2F}}, \Pi_{\mathsf{AEAD}}$, and $\mathcal{F}_{\mathsf{Com}}$ is used by Π_{AEAD}. Note that Π_{PRF} and Π_{AEAD} share the functionality $\mathcal{F}_{\mathsf{GP2PC}}$ such that the state of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ invoked by Π_{PRF} can be directly used in Π_{AEAD}. Sub-protocols Π_{PRF} and Π_{AEAD} are invoked multiple times, and we use $\Pi^{(i)}_{\mathsf{PRF}}$ (resp., $\Pi^{(i)}_{\mathsf{AEAD}}$) to denote the i-th execution of Π_{PRF} (resp., Π_{AEAD}). In protocol Π_{AuthData}, $\widetilde{\mathsf{pms}} \in \mathbb{Z}_p$ is a pre-master secret, and pms represents its bit-decomposition form. Two parties $\mathcal{P}$ and $\mathcal{V}$ need to evaluate a modulo-addition circuit to transform an additive sharing of $\widetilde{\mathsf{pms}}$ over finite field $\mathbb{Z}_p$ into that of pms over a binary field by calling $\mathcal{F}_{\mathsf{GP2PC}}$.

In the handshake and record phases, all secrets and application keys are stored by functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as its state, and they are implicitly input by both parties to these sub-protocol executions and in turn are implicitly input to $\mathcal{F}_{\mathsf{GP2PC}}$. For the sake of simplicity, we omit how state maintained by functionality $\mathcal{F}_{\mathsf{GP2PC}}$ is updated for the description of protocol Π_{AuthData}. Below, we show how $\mathcal{F}_{\mathsf{GP2PC}}$ updates state in the handshake and record phases:

$$\begin{aligned}
&\mathsf{state} = \perp \rightarrow \mathit{AddModp} \rightarrow \mathsf{state} = \mathsf{pms},\\
&\mathsf{state} = \mathsf{pms} \rightarrow \Pi^{(1)}_{\mathsf{PRF}} \rightarrow \mathsf{state} = \mathsf{ms},\\
&\mathsf{state} = \mathsf{ms} \rightarrow \Pi^{(2)}_{\mathsf{PRF}} \rightarrow \mathsf{state} = (\mathsf{ms}, \mathsf{key}_C, \mathsf{key}_S),\\
&\mathsf{state} \rightarrow \Pi^{(3)}_{\mathsf{PRF}} \rightarrow \Pi^{(1)}_{\mathsf{AEAD}} \rightarrow \Pi^{(4)}_{\mathsf{PRF}} \rightarrow \mathsf{state} = (\mathsf{key}_C, \mathsf{key}_S),\\
&\mathsf{state} \rightarrow \Pi^{(2)}_{\mathsf{AEAD}} \rightarrow \Pi^{(3)}_{\mathsf{AEAD}} \rightarrow \mathsf{state}
\end{aligned}$$

In the post-record phase, functionality $\mathcal{F}_{\mathsf{GP2PC}}$ updates state^* in a similar manner.

As in DECO [ZMM+20], 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 [ZMM+20]. In Section 4.3, we describe how to extend Π_{AuthData} to support multi-round sessions. For one-round session, $\mathcal{P}$ is *unnecessary* to decrypt the AEAD ciphertext $\textsc{enc}_R$ 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 $\textsc{enc}_R$ was received by $\mathcal{P}$ and forwarded to $\mathcal{V}$. Nevertheless, $\mathcal{P}$ and $\mathcal{V}$ still need to generate $[\![\textsc{z}_R]\!]$ and $[\![R]\!]$ by calling functionality $\mathcal{F}_{\mathsf{IZK}}$. $\mathcal{V}$ also needs to check the correctness of the GMAC tag in ciphertext $\textsc{enc}_R$ via getting $h_S = \mathsf{AES}(\mathsf{key}_S, \mathbf{0})$ and $\textsc{z}_R = \mathsf{AES}(\mathsf{key}_S, \mathsf{st}^C_d)$.

Theorem 1. *If the decisional Diffie-Hellman (DDH) assumption holds, then protocol* Π_{AuthData} *(shown in Figures 4 and 5) securely realizes functionality* $\mathcal{F}_{\mathsf{AuthData}}$ *(shown in Figure 1) 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 the above theorem in Appendix F.

4.3 Extension and Optimization of Our Main Protocol

Extend to multi-round query-response sessions. $\mathcal{P}$ and $\mathcal{V}$ can execute sub-protocol Π_{AEAD} (shown in Figures 15 and 16) 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 in these sub-protocol executions. Note that the state st^C_e is always increased for computing multiple AEAD ciphertexts following the TLS specification. This prevents $\mathcal{P}$ or $\mathcal{V}$ to forge MAC tags by using the same state to compute the tags on different ciphertexts.

If every query is independent of previous responses, $\mathcal{P}$ can *locally* decrypt the AEAD ciphertexts of all responses, after the TLS session terminates and it obtains key_S. If every query relies on previous responses, $\mathcal{P}$ has to decrypt the ciphertexts of all responses via interacting with $\mathcal{V}$. This can be done by running sub-protocol Π_{AEAD}, which is supported for type_1 = "decryption" and type_2 = "secret". During the protocol execution, Π_{AEAD} also allows $\mathcal{P}$ and $\mathcal{V}$ to verify correctness of GMAC tags in AEAD ciphertexts of responses. Therefore, in both cases, $\mathcal{P}$ can check correctness of AEAD ciphertext of every response via sending the ciphertext to $\mathcal{V}$ and then running sub-protocol Π_{AEAD} with $\mathcal{V}$, before generating a ciphertext of next query. In fact, this is unnecessary and the GMAC tags of AEAD ciphertexts of all responses can be verified *locally* by $\mathcal{P}$ after it knows key_S (see below for discussion of this optimization).

	Communication cost				WAN (100Mbps, RTT=50ms)				LAN (1Gbps, RTT=0ms)			
	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	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
Ours	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 and our protocol under LAN and WAN.**

In the case of multi-round sessions, $\mathcal{V}$ can use the same approach implied in protocol Π_{AuthData} to check correctness of all ciphertexts of multiple queries and responses in the post-record phase, and both parties are able to obtain IT-MACs of all queries and responses in a similar manner.

Optimization. We can further optimize the efficiency of protocol Π_{AuthData} by delaying the check of $\textsc{ufin}_{\mathcal{S}}$ and AEAD ciphertext $\textsc{fin}_{\mathcal{S}}$ from the handshake/record phase to the post-record phase. That is, $\mathcal{P}$ and $\mathcal{V}$ do *not* execute sub-protocol Π_{AEAD} to generate $\textsc{ufin}_{\mathcal{S}}$ and the GMAC tag used to check $\textsc{fin}_{\mathcal{S}}$. Instead, $\mathcal{P}$ can *locally* check their correctness after it obtains master secret ms. $\mathcal{V}$ checks correctness of $\textsc{ufin}_{\mathcal{S}}$ by calling the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ with $\mathcal{P}$, and checks correctness of $\textsc{fin}_{\mathcal{S}}$ in the way shown in the post-record phase of main protocol Π_{AuthData}. This has *no* impact on privacy and integrity, as this optimization only delays the check. If these values are incorrect, $\mathcal{P}$ and $\mathcal{V}$ abort. 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 $\mathcal{V}$ delay the check of correctness of all AEAD ciphertexts of responses from the record phase into the post-record phase by using the approach shown in protocol Π_{AuthData}. This optimization allows us to reduce communication rounds and improve performance.

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 [WMK16] library for basic building blocks like oblivious transfer [KOS15, YWL^{+}20], garbled circuits [ZRE15, GKWY20] and interactive ZK [YSWW21]. We leave it as future work to incorporate the recent three-halves construction [RR21] to further reduce the overhead of our protocol.

All benchmarks are performed over AWS m5.large instances, with 2 vCPUs and 8GB 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 API 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. We use one thread for all running time and tc to manually control the network bandwidth and roundtrip latency to desired levels. When we report running time and communication, they include all preprocessing and setup cost.

5.2 Scalability of Our Protocol

Performance of the main protocol. In Figure 6, we show the performance of our main protocol under different bandwidth and latency settings 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

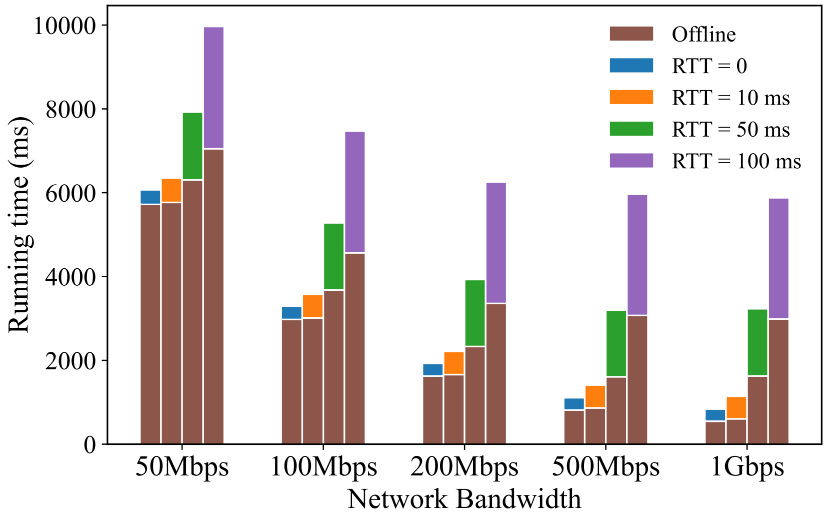

Figure 6: **Performance of our protocol under different network bandwidths and latency.** The length of query and response is 2KB.

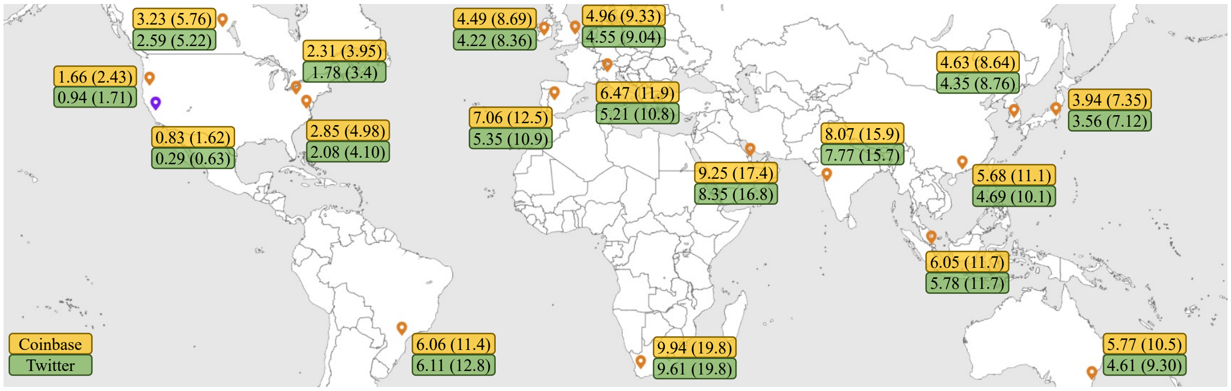

Figure 7: **Online and total performance of accessing Coinbase and Twitter servers with globally distributed provers.** All numbers are reported in seconds in the form of "online time (total time)". The verifier is fixed at California. The server is hosted by Coinbase/Twitter, which may have mirrors in various locations.

example, under a realistic network with 200Mbps bandwidth and 50ms latency, the total cost is under four seconds while the online phase is less than two seconds.

We can also see that the online cost 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 the garbled circuit, 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. 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 oblivious transfer [YWL+20] 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

	WAN (100 Mbps, RTT=50ms)				LAN (1 Gbps, RTT=0ms)			
	256 B	512 B	1 KB	2 KB	256 B	512 B	1 KB	2 KB
Conv	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.**

computing the TLS session keys and 4 pairs of AES-GCM ciphertext/tags. When computing the GCM 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 [HK21] 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 instance. 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 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. We observe that in both cases, 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 37ms to convert an additional kilobyte of payload to Pedersen commitment under LAN and roughly 67ms per KB under WAN. The basetime in WAN is higher due to the higher cost by the latency.

5.3 Global-Scale Benchmarks

We also integrate our protocol to access real-world web servers and test the online performance, as shown in Figure 7. Specifically, we utilize provided APIs to query Coinbase and Twitter servers.

- **Coinbase API**: we benchmark fetching the balance of BTC using prover's API secret [coi]. It has a query of size 426 bytes and response of size 5701 bytes. Our protocol communicates 17.6 MB offline and 0.9 MB online.
- **Twitter API**: we benchmark using prover's credential token to retrieve the number of followers [twi]. 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 various locations 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. We conduct tests with provers deployed in 18 different cities. The time required for the process ranges from 0.3 seconds to 10 seconds, depending on the round-trip time between the prover and verifier, which aligns with our expectations.

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 enhance the overall performance of the system. By strategically locating the verifiers closer to the provers, we can achieve reduced latency and optimize the efficiency of the process.

Acknowledgements

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), the National Key Research and Development Program of China (Grant No. 2018YFA0704701), and the Major Program of Guangdong Basic and Applied Research (Grant No. 2019B030302008). Yu Yu also acknowledges the support from the XPLORER PRIZE. Xiao Wang is supported by DARPA under Contract No. HR001120C0087, NSF awards #2016240, #2236819, 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

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE Computer Society Press, May 2013.

[BCG$^+$19a] 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 Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

[BCG$^+$19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[BCG$^+$20] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 387–416. Springer, Heidelberg, August 2020.

[BCG$^+$22] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 603–633. Springer, Heidelberg, August 2022.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[BLN$^+$21] 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. *Journal of Cryptology*, 34(3):34, July 2021.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[BMR16] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for Boolean and arithmetic circuits. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 565–577. ACM Press, October 2016.

[BMRS21] 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 Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Heidelberg.

[Bra13] 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 Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 441–463. Springer, Heidelberg, December 2013.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

[CDE$^+$18] 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 Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018.

[CDH$^+$16] 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.

[CFF$^+$21] 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. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2021.

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2075–2092. ACM Press, November 2019.

[CHM$^+$20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the “free-XOR” technique. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, Heidelberg, March 2012.

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

[CRR21] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 502–534, Virtual Event, August 2021. Springer, Heidelberg.

[DILO22] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Authenticated garbling from simple correlations. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 57–87. Springer, Heidelberg, August 2022.

[DIO21] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In *2nd Conference on Information-Theoretic Cryptography*, 2021.

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 167–187. Springer, Heidelberg, August 2017.

[DR08] 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.

[EGK$^+$20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Heidelberg, August 2020.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735. Springer, Heidelberg, November / December 2015.

[Gil99] Niv Gilboa. Two party RSA key generation. In Michael J. Wiener, editor, *CRYPTO’99*, volume 1666 of *LNCS*, pages 116–129. Springer, Heidelberg, August 1999.

[GKMN21] François Garillot, Yashvanth Kondi, Payman Mohassel, and Valeria Nikolaenko. Threshold Schnorr with stateless deterministic signing from standard assumptions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 127–156, Virtual Event, August 2021. Springer, Heidelberg.

[GKWY20] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy*, pages 825–841. IEEE Computer Society Press, May 2020.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[Har12] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. https://www.rfc-editor.org/info/rfc6749.

[HK21] David Heath and Vladimir Kolesnikov. One hot garbling. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 574–593. ACM Press, November 2021.

[HSS20] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. *Journal of Cryptology*, 33(4):1732–1786, October 2020.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[IT21] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021. https://www.rfc-editor.org/info/rfc9000.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[KE10] 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.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, August 2010.

[KRRW18] Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 365–391. Springer, Heidelberg, August 2018.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 495–512. Springer, Heidelberg, August 2014.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, Heidelberg, May 2007.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, Heidelberg, August 2015.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, November 2019.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, Heidelberg, March 2009.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Heidelberg, August 1992.

[Res18] E. Rescorla. The transport layer security (TLS) protocol version 1.3. RFC 8446, August 2018. https://www.rfc-editor.org/rfc/rfc8446.txt.

[Roy22] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Heidelberg, August 2022.

[RR21] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 94–124, Virtual Event, August 2021. Springer, Heidelberg.

[SBJ$^+$14] Nat Sakimura, John Bradley, Michael B. Jones, Breno de Medeiros, and Chuck Mortimore. OpenID Connect Core 1.0 incorporating errata set 1, 2014.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.

[sS11] abhi shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 386–405. Springer, Heidelberg, May 2011.

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

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

[TLS23] TLSNotary. Proof of data authenticity. https://docs.tlsnotary.org, Access at 2023. Source code is available at https://github.com/tlsnotary/tlsn.

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

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient Multi-Party computation toolkit. https://github.com/emp-toolkit, 2016.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 21–37. ACM Press, October / November 2017.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021.

[WYX$^+$21] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 501–518. USENIX Association, August 2021.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021.

[YWL$^+$20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1607–1626. ACM Press, November 2020.

[YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In Jay Ligatti, Xinming Ou, Jonathan Katz,

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

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

Figure 8: **Functionality for oblivious transfer.**

and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1627–1646. ACM Press, November 2020.

[ZMM+20] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1919–1938. ACM Press, November 2020.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.

A More Preliminaries

A.1 Security Model and Functionalities

We use the standard ideal/real paradigm [Can00, Gol04] 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 $\mathcal{G}$.

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 8. 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 [KOS15, Roy22] or the PCG-like protocols [BCG+19b, BCG+19a, YWL+20].

OLE. 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 $\mathcal{F}_{\mathsf{OLEe}}$

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

- Upon receiving (ole, x) from a sender $\mathsf{P_A}$ and (ole, y) from a receiver $\mathsf{P_B}$ where $x, y \in \mathbb{F}$, execute as follows:

  1. If $\mathsf{P_A}$ is honest, sample $z_A \leftarrow \mathbb{F}$. Otherwise, receive $z_A \in \mathbb{F}$ from the adversary.
  2. If $\mathsf{P_A}$ is malicious, receive a vector $\boldsymbol{e} \in (\mathbb{F})^m$ from the adversary, and compute an error $e' := (\boldsymbol{g} * \boldsymbol{e}) \odot \boldsymbol{y} \in \mathbb{F}$ where $\boldsymbol{y} = \boldsymbol{g}^{-1}(y)$, $*$ is a component-wise product and $\odot$ denotes inner product.
  3. If $\mathsf{P_B}$ is honest, compute $z_B := x \cdot y - z_A + e' \in \mathbb{F}$ (where e' is set as 0 if $\mathsf{P_A}$ is also honest). Otherwise, receive $z_B \in \mathbb{F}$ from the adversary, and recompute $z_A := x \cdot y - z_B \in \mathbb{F}$.

- Output z_A to $\mathsf{P_A}$ and z_B to $\mathsf{P_B}$.

Figure 9: **Functionality for OLE with errors.**

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

This functionality runs with two parties $\mathsf{P_A}$ and $\mathsf{P_B}$, and operates as follows.

Commit: Upon receiving $(\mathsf{commit}, \mathsf{cid}, x)$ from $\mathsf{P_A}$, store (cid, x) and output $(\mathsf{committed}, \mathsf{cid})$ to $\mathsf{P_B}$. Ignore any subsequent (commit) command with the same cid.

Open: Upon receiving $(\mathsf{open}, \mathsf{cid})$ from $\mathsf{P_A}$, if (cid, x) was previously stored, then output $(\mathsf{opened}, \mathsf{cid}, x)$ to $\mathsf{P_B}$.

Linear combination: Upon receiving $(\mathsf{lincomb}, \mathsf{cid}', \mathsf{cid}_1, \ldots, \mathsf{cid}_n, c_0, c_1, \ldots, c_n)$ from $\mathsf{P_A}$ and $\mathsf{P_B}$, if (cid_i, x_i) for all $i \in [1, n]$ are previously stored and c_i for all $i \in [0, n]$, compute $y := \sum_{i=1}^{n} c_i \cdot x_i + c_0$, store (cid', y) and send $(\mathsf{done}, \mathsf{cid}', \{\mathsf{cid}_i\}_{i=1}^{n}, \{c_i\}_{i=0}^{n})$ to both parties.

Figure 10: **Functionality for homomorphic commitments.**

Functionality for OLE with errors is shown in Figure 9. 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 $\boldsymbol{g} \odot \boldsymbol{x} = \sum_{i=1}^{m} x_i \cdot g^{i-1} \in \mathbb{F}$ where $\odot$ denotes inner product. 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 $\boldsymbol{g} \odot \boldsymbol{g}^{-1}(x) = x$. Following previous work (e.g., [BCG$^+$20]), we allow a corrupted party to choose its output. If a sender $\mathsf{P_A}$ 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 on the input y of a receiver $\mathsf{P_B}$. Finally, the error e' is added into the output z_B of $\mathsf{P_B}$. The introduction of errors is asymmetric, i.e., $\mathsf{P_B}$ is *not* allowed to add an error into the output of $\mathsf{P_A}$. This model the asymmetric security of the COT-based protocol [Gil99, KOS16] that securely realizes functionality $\mathcal{F}_{\mathsf{OLEe}}$. This protocol allows us to obtain fast computation, where the communication of OLEe is only a small part of communication of our protocol.

Commitment. Our protocol adopts an additively homomorphic commitment scheme, which is modeled in the functionality $\mathcal{F}_{\mathsf{HCom}}$ shown in Figure 10. We always assume that the message space of values to be committed is a finite field $\mathbb{F}$, and denote by $\mathcal{F}_{\mathsf{HCom}}[\mathbb{F}]$ the functionality with message space $\mathbb{F}$. In this case, the $\mathsf{lincomb}$ command is well-defined where all operations are defined over $\mathbb{F}$. We need that such commitment scheme is *non-interactive*. For example, the Pedersen commitment scheme [Ped92] satisfies the requirement. To realize functionality $\mathcal{F}_{\mathsf{HCom}}$, we need that Pedersen commitment is equipped with a non-interactive ZK proof (or proved under

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

This functionality has all the features of $\mathcal{F}_{\mathsf{GP2PC}}$ (shown in Figure 3), and also involves the following commands.

Initialize. Upon receiving (init) from a prover $\mathcal{P}$ and (init, Δ) from a verifier $\mathcal{V}$ where $\Delta \in \{0,1\}^\lambda$, store Δ and ignore all subsequent (init) commands.

Input authentication. Upon receiving $(\mathsf{authinput}, \mathsf{id}, w)$ from $\mathcal{P}$ and $(\mathsf{authinput}, \mathsf{id})$ from $\mathcal{V}$, where $w \in \{0,1\}$ and id is a fresh identifier, run $\mathsf{Auth}(w)$ so that the parties obtain $[\![w]\!]$ and store $(\mathsf{id}, [\![w]\!])$.

Prove circuits. Upon receiving $(\mathsf{zkauth}, \mathcal{C}, \mathbf{id}^{in}, \mathbf{id}^{out})$ from $\mathcal{P}$ and $\mathcal{V}$, where $\mathcal{C} : \{0,1\}^m \to \{0,1\}^n$ is a Boolean circuit and id_i^{in} for all $i \in [1,m]$ are present in memory, retrieve $(\mathsf{id}_i^{in}, [\![x_i]\!])$ for $i \in [1,m]$, and compute $(y_1, \ldots, y_n) := \mathcal{C}(x_1, \ldots, x_m)$. For $i \in [1,n]$, run $\mathsf{Auth}(y_i)$ so that the parties obtain $[\![y_i]\!]$ and store $(\mathsf{id}_i^{out}, [\![y_i]\!])$.

Check. Upon receiving $(\mathsf{check}, \mathsf{id})$ from $\mathcal{P}$ and $\mathcal{V}$, if $(\mathsf{id}, [\![y]\!])$ was previously stored, then send true to $\mathcal{V}$ if $y = 0$ or false otherwise.

Macro $\mathsf{Auth}(x)$ (this is an internal subroutine only)

If Δ was previously stored, do the following:

- If $\mathcal{V}$ is honest, sample $\mathsf{K}[x] \leftarrow \{0,1\}^\lambda$. Otherwise, receive $\mathsf{K}[x] \in \{0,1\}^\lambda$ from the adversary.
- If $\mathcal{P}$ is honest, compute $\mathsf{M}[x] := \mathsf{K}[x] \oplus x\Delta$. Otherwise, receive $\mathsf{M}[x] \in \{0,1\}^\lambda$ from the adversary and recompute $\mathsf{K}[x] := \mathsf{M}[x] \oplus x\Delta$.
- Send $(x, \mathsf{M}[x])$ to $\mathcal{P}$ and $\mathsf{K}[x]$ to $\mathcal{V}$.

Figure 11: **Functionality for interactive ZK based on IT-MACs.**

generic group model [Sho97]).

In addition, our protocol also needs to call the standard commitment functionality (denoted by $\mathcal{F}_{\mathsf{Com}}$) without homomorphic properties. This functionality is the same as that shown in Figure 10 except that the $(\mathsf{lincomb})$ command is removed and message space is $\{0,1\}^*$ rather than $\mathbb{F}$. Functionality $\mathcal{F}_{\mathsf{Com}}$ can be securely realized by defining $\mathsf{H}(m, r)$ as a commitment on a message m, where H is a random oracle and r is a randomness.

A.2 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 $\mathbb{F}$. 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 secret x is denoted by $[x]_p$ for former and $[x]_{2^{128}}$ for latter.

A.3 Interactive ZK Proofs based on IT-MACs

Based on IT-MACs, a family of interactive zero-knowledge (IZK) proofs with fast prover time and a small memory footprint was recently proposed [WYKW21, DIO21, BMRS21].Our protocol will use an IZK proof to prove satisfiability of a Boolean circuit. Such ZK proofs can commit to a witness using IT-MACs, evaluate the circuit such that all wire values are committed with IT-MACs, and then enable the prover and verifier to obtain IT-MACs on output values.

In Figure 11, we define an ideal functionality $\mathcal{F}_{\mathsf{IZK}}$ that models security of the recent IZK proofs for Boolean circuits based on IT-MACs. $\mathcal{F}_{\mathsf{IZK}}$ is the simplification of the ideal functionality defined in the previous work [WYX$^+$21]. Functionality $\mathcal{F}_{\mathsf{IZK}}$ is able to be efficiently realized by the recent IT-MACs-based IZK protocol such as [YSWW21]. At the beginning of execution, a verifier $\mathcal{V}$ samples a uniform global key $\Delta \in \{0,1\}^\lambda$ to initialize functionality $\mathcal{F}_{\mathsf{IZK}}$. Then, a prover $\mathcal{P}$ and the verifier $\mathcal{V}$ can authenticate witnesses via the (input) command. Following previous work [CDE$^+$18, WYX$^+$21], a macro Auth is defined to generate IT-MACs, and allows one corrupted party to choose its output from Auth. $\mathcal{P}$ can convince $\mathcal{V}$ that a circuit output is computed correctly by calling the (zkauth) command in which all output bits are authenticated with IT-MACs. To check that $y = y'$ for an IT-MAC $[\![y]\!]$ and public value y', $\mathcal{P}$ and $\mathcal{V}$ can call the (prove) command to compute $[\![y]\!] - y'$, and then call the (check) command to verify that $y - y' = 0$.

B Sub-Protocols for TLS Building Blocks

We include a graphically illustration of the TLS protocol in Figure 2 and full details in Figure 12.

B.1 Sub-Protocol for Conversion of Sharings

In Figure 13, we present a sub-protocol in the $\mathcal{F}_{\mathsf{OLEe}}$-hybrid model to convert additive sharings of elliptic-curve (EC) points to that of x-coordinates. Let $EC(\mathbb{Z}_p)$ be an elliptic curve defined over a finite field $\mathbb{Z}_p$ for a prime p, where $\mathbb{Z}_p$ is the base field that coordinates locate in. We abuse the notation, and still use $+$ denote addition operation over $EC(\mathbb{Z}_p)$. Nevertheless, we note that addition operation over $EC(\mathbb{Z}_p)$ is different from that over $\mathbb{Z}_p$. For two EC points $Z_1 = (x_1, y_1)$ and $Z_2 = (x_2, y_2)$ with $x_1 \neq x_2$, the x-coordinate of their addition $z = \mathsf{F}_x(Z_1 + Z_2)$ is computed as $z = \eta^2 - x_1 - x_2 \in \mathbb{Z}_p$ where $\eta = (y_2 - y_1)/(x_2 - x_1) \in \mathbb{Z}_p$. Similar to the conversion protocol in DECO [ZMM$^+$20], this sub-protocol uses OLE correlations to compute the coordinate z. Compared to the protocol [ZMM$^+$20], our protocol has two different points: one is that we only adopt OLE with errors (instead of fully secure OLE); the other is that we divide it into the preprocessing phase and online phase to obtain fast online efficiency.

B.2 Sub-Protocol for Computing HMAC-PRF

In Figure 14, we present the details of a concretely efficient 2PC protocol to securely compute the HMAC-based pseudorandom function PRF_ℓ defined in Section 2.1, where ℓ is the output length of PRF. When applying this protocol into our main protocol shown in Section 4.2, sub-protocol Π_{PRF} (Figure 14) will be used in three different cases that are distinguished by a label type.

- If type = "secret", Π_{PRF} is used to generate a master secret ms from a pre-master secret pms, where the secrets are stored by functionality $\mathcal{F}_{\mathsf{GP2PC}}$.
- If type = "open", Π_{PRF} is used to generate and open unencrypted finished messages $\textsc{ufin}_C$ and $\textsc{ufin}_S$ and to check the correctness of $\textsc{ufin}_C$ and $\textsc{ufin}_S$.

Protocol TLS 1.2

Inputs. A client $\mathcal{C}$ and a server $\mathcal{S}$ hold the following inputs:

- *Personal inputs:* $\mathcal{C}$ has a query template Query and a private input α for Query. $\mathcal{S}$ holds a secret key sk_S and a certification cert_S involving a public key pk_S.
- *Common inputs:* The common inputs except for F_x are chosen by $\mathcal{C}$ or $\mathcal{S}$ in the handshake phase.
  - Let F_x be a function mapping an elliptic-curve point to its x-coordinate.
  - Let H be a cryptographic hash function and PRF be a pseudorandom function.
  - $\mathsf{SIG} = (\mathsf{Sign}, \mathsf{Verify})$ is a signature scheme defined by cert_S, where the key-generation algorithm is omitted, Sign is the signing algorithm used to generate signatures, and Verify is the verification algorithm that outputs 0 (reject) or 1 (accept).
  - $(\mathbb{G}, p, q, G)$ is an elliptic-curve group, where p is a prime defining the base field that coordinates locate in and G is a generator with a prime order q.
  - $\mathsf{stE} = (\mathsf{Enc}, \mathsf{Dec})$ is a stateful AEAD scheme, where the key-generation algorithm is omitted.

Handshake protocol execution.

1. *Client request:* $\mathcal{C}$ samples $r_C \leftarrow \{0,1\}^{256}$ and sends $\textsc{req}_C := r_C$ to $\mathcal{S}$.
2. *Server response:* $\mathcal{S}$ performs the following steps:
   (a) Sample $r_S \leftarrow \{0,1\}^{256}$ and $t_S \leftarrow \mathbb{Z}_q$, and compute $T_S := t_S \cdot G$.
   (b) Run $\sigma_S \leftarrow \mathsf{Sign}(\mathsf{sk}_S, r_C \| r_S \| T_S)$, and then send $\textsc{res}_S := (r_S, T_S, \mathsf{cert}_S, \sigma_S)$ to $\mathcal{C}$.
3. *Client response:* If cert_S is invalid or $\mathsf{Verify}(\mathsf{pk}_S, r_C \| r_S \| T_S, \sigma_S) = 0$, $\mathcal{C}$ aborts. Otherwise, $\mathcal{C}$ samples $t_C \leftarrow \mathbb{Z}_q$ and computes $T_C := t_C \cdot G$, and then sends $\textsc{res}_C := T_C$ to $\mathcal{S}$. Both parties compute:
   (a) $\mathcal{C}$ computes $\mathsf{pms} := \mathsf{F}_x(t_C \cdot T_S) \in \mathbb{Z}_p$, and $\mathcal{S}$ computes $\mathsf{pms} := \mathsf{F}_x(t_S \cdot T_C) \in \mathbb{Z}_p$.
   (b) $\mathcal{C}$ and $\mathcal{S}$ compute $\mathsf{ms} := \mathsf{PRF}_{384}(\mathsf{pms}, \text{“master secret”}, r_C \| r_S) \in \{0,1\}^{384}$.
   (c) They compute $(\mathsf{key}_C, IV_C, \mathsf{key}_S, IV_S) := \mathsf{PRF}_{448}(\mathsf{ms}, \text{“key expansion”}, r_S \| r_C) \in \{0,1\}^{448}$ with $\mathsf{key}_C, \mathsf{key}_S \in \{0,1\}^{128}$ and $IV_C, IV_S \in \{0,1\}^{96}$.
4. *Client finished:* Let $\textsc{h}_C$ be a header specifying the sequence number, version and length of a plaintext, and ℓ_C be the target ciphertext length. $\mathcal{C}$ performs the following:
   (a) Compute $\tau_C := \mathsf{H}(\textsc{creq} \| \textsc{sres} \| \textsc{cres})$ and $\textsc{ufin}_C := \mathsf{PRF}_{96}(\mathsf{ms}, \text{“client finished”}, \tau_C) \in \{0,1\}^{96}$.
   (b) Initialize $(\mathsf{st}_e^C, \mathsf{st}_d^C) := (IV_C, IV_S)$, and run $\textsc{fin}_C \leftarrow \mathsf{stE.Enc}(\mathsf{key}_C, \ell_C, \textsc{h}_C, \textsc{ufin}_C, \mathsf{st}_e^C)$. Then, send $(\textsc{h}_C, \textsc{fin}_C)$ to $\mathcal{S}$.

   $\mathcal{S}$ initializes $(\mathsf{st}_e^S, \mathsf{st}_d^S) := (IV_S, IV_C)$, and runs $\textsc{ufin}_C \leftarrow \mathsf{stE.Dec}(\mathsf{key}_C, \textsc{h}_C, \textsc{fin}_C, \mathsf{st}_d^S)$. Then, $\mathcal{S}$ checks the validity of $\textsc{ufin}_C$ using ms, and aborts if the check fails.
5. *Server finished:* Let $\textsc{h}_S$ be a header and ℓ_S be the target ciphertext length. $\mathcal{S}$ does the following:
   (a) Compute $\tau_S := \mathsf{H}(\textsc{req}_C \| \textsc{res}_S \| \textsc{res}_C \| \textsc{ufin}_C)$ and $\textsc{ufin}_S := \mathsf{PRF}_{96}(\mathsf{ms}, \text{“server finished”}, \tau_S) \in \{0,1\}^{96}$.
   (b) Run $\textsc{fin}_S \leftarrow \mathsf{stE.Enc}(\mathsf{key}_S, \ell_S, \textsc{h}_S, \textsc{ufin}_S, \mathsf{st}_e^S)$, and then send $(\textsc{h}_S, \textsc{fin}_S)$ to $\mathcal{C}$.

   $\mathcal{C}$ runs $\textsc{ufin}_S \leftarrow \mathsf{stE.Dec}(\mathsf{key}_S, \textsc{h}_S, \textsc{fin}_S, \mathsf{st}_d^C)$ and checks $\textsc{ufin}_S$ using ms, and aborts if the check fails.

Record protocol execution with one-round session.

6. *Client query:* Let $\textsc{h}_Q$ be a header and ℓ_Q be the target ciphertext length. $\mathcal{C}$ and $\mathcal{S}$ execute as follows:
   (a) $\mathcal{C}$ runs $Q := \mathsf{Query}(\alpha)$ and $\textsc{enc}_Q \leftarrow \mathsf{stE.Enc}(\mathsf{key}_C, \ell_Q, \textsc{h}_Q, Q, \mathsf{st}_e^C)$, and sends $(\textsc{h}_Q, \textsc{enc}_Q)$ to $\mathcal{S}$.
   (b) $\mathcal{S}$ runs $Q \leftarrow \mathsf{stE.Dec}(\mathsf{key}_C, \textsc{h}_Q, \textsc{enc}_Q, \mathsf{st}_d^S)$, and generates R according to Q.
7. *Server response:* Let $\textsc{h}_R$ be a header and ℓ_R be the target ciphertext length. $\mathcal{C}$ and $\mathcal{S}$ do the following:
   (a) $\mathcal{S}$ runs $\textsc{enc}_R \leftarrow \mathsf{stE.Enc}(\mathsf{key}_S, \ell_R, \textsc{h}_R, R, \mathsf{st}_e^S)$, and then sends $(\textsc{h}_R, \textsc{enc}_R)$ to $\mathcal{S}$.
   (b) $\mathcal{C}$ gets $R \leftarrow \mathsf{stE.Dec}(\mathsf{key}_S, \textsc{h}_R, \textsc{enc}_R, \mathsf{st}_d^C)$.

Figure 12: **Handshake and record protocols for TLS 1.2.**

Protocol Π_{E2F}

Inputs. $\mathcal{P}$ holds an EC point $Z_1 = (x_1, y_1)$, and $\mathcal{V}$ has an EC point $Z_2 = (x_2, y_2)$, where these coordinates are defined over $\mathbb{Z}_p$.

Preprocessing phase. Before launching a TLS session, $\mathcal{P}$ and $\mathcal{V}$ execute the following preprocessing:

1. $\mathcal{P}$ and $\mathcal{V}$ sample $a_1, b_1, b'_1, r_1 \leftarrow \mathbb{Z}_p$ and $a_2, b_2, b'_2, r_2 \leftarrow \mathbb{Z}_p$, respectively. Then, both parties define additive sharings $[a]_p = (a_1, a_2)$, $[b]_p = (b_1, b_2)$, $[b']_p = (b'_1, b'_2)$, $[r]_p = (r_1, r_2)$.
2. $\mathcal{P}$ (as a sender) and $\mathcal{V}$ (as a receiver) call functionality $\mathcal{F}_{\mathsf{OLEe}}$ on respective input $(a_1, b_1, a_1, b'_1, r_1)$ and $(b_2, a_2, b'_2, a_2, r_2)$ to obtain additive sharings $[a_1 b_2]_p, [a_2 b_1]_p, [a_1 b'_2]_p, [a_2 b'_1]_p$ and $[r_1 r_2]_p$.
3. Both parties locally compute additive sharings $[c]_p := [a_1 b_1]_p + [a_1 b_2]_p + [a_2 b_1]_p + [a_2 b_2]_p$ and $[c']_p := [a_1 b'_1]_p + [a_1 b'_2]_p + [a_2 b'_1]_p + [a_2 b'_2]_p$, where $c = a \cdot b \in \mathbb{Z}_p$ and $c' = a \cdot b' \in \mathbb{Z}_p$.
4. $\mathcal{P}$ and $\mathcal{V}$ locally compute $[r^2]_p := [r_1^2]_p + 2 \cdot [r_1 r_2]_p + [r_2^2]_p$.

Handshake phase. When the inputs (x_1, y_1) and (x_2, y_2) are known, $\mathcal{P}$ and $\mathcal{V}$ do the following:

5. Both parties define $[x_2 - x_1]_p = (-x_1, x_2)$, and compute $[w]_p = [(x_2 - x_1) \cdot a]_p$ as follows:
   (a) $\epsilon_1 \leftarrow \mathsf{Open}([x_2 - x_1]_p - [b]_p)$.
   (b) $[w]_p := \epsilon_1 \cdot [a]_p + [c]_p$.
6. The parties run $w \leftarrow \mathsf{Open}([w]_p)$ and abort if $w = 0$. Both parties define $[y_2 - y_1]_p = (-y_1, y_2)$, and then compute $[\eta]_p = [(y_2 - y_1)/(x_2 - x_1)]_p = w^{-1} \cdot [(y_2 - y_1) \cdot a]_p$ as follows:
   (a) $\epsilon_2 \leftarrow \mathsf{Open}([y_2 - y_1]_p - [b']_p)$.
   (b) $[\eta]_p := w^{-1} \cdot (\epsilon_2 \cdot [a]_p + [c']_p)$.
7. Two parties compute $[z]_p = [\eta^2 - x_1 - x_2]_p$ as follows:
   (a) $\epsilon_3 \leftarrow \mathsf{Open}([\eta]_p - [r]_p)$.
   (b) $[z]_p := \epsilon_3^2 + 2\epsilon_3 \cdot [r]_p + [r^2]_p - [x_1]_p - [x_2]_p$.
8. $\mathcal{P}$ and $\mathcal{V}$ output an additive sharing $[z]_p$ with $z = \mathsf{F}_x(Z_1 + Z_2)$.

Figure 13: **Protocol for conversion of additive secret sharings from EC points to x-coordinates in the $\mathcal{F}_{\mathsf{OLEe}}$-hybrid model.**

- If type = "partial open", Π_{PRF} is used to generate a tuple $(\mathsf{key}_C, IV_C, \mathsf{key}_S, IV_S)$ and open (IV_C, IV_S) where $\mathsf{key}_C, \mathsf{key}_S$ are two application keys and IV_C, IV_S are public initial vectors. In this case, Π_{PRF} is also used to check the correctness of IV_C, IV_S.

As stated in Section 3.2, we allow $\mathcal{P}$ and $\mathcal{V}$ to reveal some intermediate values, which is secure in the random oracle model. In this case, we split the Boolean circuit of computing PRF_ℓ into three sub-circuits $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$. This allows us to formally describe how to reveal the values. In the post-record phase, the corresponding verification circuits are proved to check the correctness of the values opened. Since the values have been revealed, we can utilize the public values to define circuits $\mathcal{C}'_{2,i}$ for $i \in [1, n]$ and $\mathcal{C}'_3$, which are obtained by transforming a part of inputs for circuits $\mathcal{C}_2, \mathcal{C}_3$ into public values.

For the sake of simplicity, we omit how state maintained by functionality $\mathcal{F}_{\mathsf{GP2PC}}$ is updated in the description of sub-protocol Π_{PRF}. Below, we show how $\mathcal{F}_{\mathsf{GP2PC}}$ updates state in the handshake and record phases:

- If type = "secret",

$$\mathsf{state} = \mathsf{pms} \rightarrow \mathcal{C}_1 \rightarrow \mathsf{state} = IV_2,$$
$$\mathsf{state} = IV_2 \rightarrow \mathcal{C}_2 \rightarrow \mathsf{state} = IV_2,$$

Protocol Π_{PRF}

Inputs. $\mathcal{P}$ and $\mathcal{V}$ input two bit strings *label* and *msg* as well as a label $\mathsf{type} \in \{$"open", "partial open", "secret"$\}$. Let $n = \lceil \ell/256 \rceil$ and $m = \ell - 256 \cdot (n-1)$. Suppose that functionality $\mathcal{F}_{\mathsf{GP2PC}}$ stores $\mathsf{secret} \in \{\mathsf{pms}, \mathsf{ms}\}$ that is packed into 512 bits with zero.

Definition of circuits. $\mathcal{P}$ and $\mathcal{V}$ define the following Boolean circuits:

- $\mathcal{C}_1(\mathsf{secret})$ inputs $\mathsf{secret} \in \{0,1\}^{512}$, and outputs $IV_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret} \oplus \text{ipad})$ as well as $IV_2 = f_{\mathsf{H}}(IV_0, \mathsf{secret} \oplus \text{opad})$, where f_{H} is the compression function of H and IV_0 is a fixed initial vector.
- $\mathcal{C}_2(IV_2, W_i)$ takes as input $IV_2, W_i \in \{0,1\}^{256}$, and outputs $M_i = f_{\mathsf{H}}(IV_2, W_i)$.
- $\mathcal{C}_3(IV_2, X_1, \ldots, X_n)$ takes as input $IV_2, X_1, \ldots, X_n \in \{0,1\}^{256}$, and then outputs $\mathsf{der} = \big(f_{\mathsf{H}}(IV_2, X_1), \ldots, f_{\mathsf{H}}(IV_2, X_{n-1}), \mathsf{Trunc}_m(f_{\mathsf{H}}(IV_2, X_n))\big) \in \{0,1\}^{\ell}$.

Let $\mathcal{C}'_{2,i}(IV_2)$ be the Boolean circuit that outputs $\mathcal{C}_2(IV_2, W_i)$ for a public value W_i, and $\mathcal{C}'_3(IV_2)$ be the Boolean circuit that outputs $\mathcal{C}_3(IV_2, X_1, \ldots, X_n)$ for public values $X_1, \ldots, X_n$.

Handshake phase. When the inputs are known, $\mathcal{P}$ and $\mathcal{V}$ do the following:

1. Both parties call the (eval) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on state secret and circuit $\mathcal{C}_1$ to compute IV_1 and IV_2. Then, $\mathcal{P}$ and $\mathcal{V}$ call the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ to open IV_1 to both parties.
2. Let $M_0 = \mathit{label} \| \mathit{msg}$. From $i = 1$ to n, both parties compute $W_i := f_{\mathsf{H}}(IV_1, M_{i-1})$, and call the (eval) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on state IV_2, circuit $\mathcal{C}_2$ and $\mathcal{P}$'s input W_i to compute $M_i = f_{\mathsf{H}}(IV_2, W_i)$. Then, $\mathcal{P}$ and $\mathcal{V}$ call the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ such that M_i for all $i \in [1, n]$ are opened to both parties.
3. For $i \in [1, n]$, both parties compute $X_i := f_{\mathsf{H}}(IV_1, M_i \| \mathit{label} \| \mathit{msg})$. Then, the parties call the (eval) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on circuit $\mathcal{C}_3$ and $\mathcal{P}$'s input $(X_1, \ldots, X_n)$ to compute the output der.
4. If $\mathsf{type} =$ "open", $\mathcal{P}$ and $\mathcal{V}$ call the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ to open der. If $\mathsf{type} =$ "partial open", both parties call (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ to open (IV_C, IV_S), where $\mathsf{der} = (\mathsf{key}_C, IV_C, \mathsf{key}_S, IV_S)$.

Post-record phase. Functionality $\mathcal{F}_{\mathsf{GP2PC}}$ stores secret^* that is identical to secret in the honest case, where $\mathcal{P}$ knows secret^* in this phase. $\mathcal{P}$ and $\mathcal{V}$ do the following, and $\mathcal{V}$ aborts if any check fails.

5. Both parties call the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on state secret^* and circuit $\mathcal{C}_1$ to generate $IV_2^* = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \text{opad})$ and check $IV_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \text{ipad})$.
6. For $i \in [1, n]$, both parties call the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on state secret^* and circuit $\mathcal{C}'_{2,i}$ to check that $M_i = f_{\mathsf{H}}(IV_2^*, W_i)$ for public value W_i.
7. Both parties call the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on state secret^* and circuit $\mathcal{C}'_3$ to generate $\mathsf{der}^* = \big(f_{\mathsf{H}}(IV_2^*, X_1), \ldots, f_{\mathsf{H}}(IV_2^*, X_{n-1}), \mathsf{Trunc}_m(f_{\mathsf{H}}(IV_2^*, X_n))\big)$ for public values $X_1, \ldots, X_n$. If $\mathsf{type} =$ "open", then $\mathcal{F}_{\mathsf{GP2PC}}$ checks that $\mathsf{der}^* = \mathsf{der}$. If $\mathsf{type} =$ "partial open", then $\mathsf{der}^* = (\mathsf{key}_C^*, IV_C^*, \mathsf{key}_S^*, IV_S^*)$ and $\mathcal{F}_{\mathsf{GP2PC}}$ checks that $IV_C^* = IV_C$ and $IV_S^* = IV_S$.

Figure 14: **Protocol for securely computing HMAC-based PRF in the $\mathcal{F}_{\mathsf{GP2PC}}$-hybrid model.**

$$\mathsf{state} = IV_2 \rightarrow \mathcal{C}_3 \rightarrow \mathsf{state} = \mathsf{ms}.$$

- If $\mathsf{type} =$ "partial open",

$$\begin{aligned}
&\mathsf{state} = \mathsf{ms} \rightarrow \mathcal{C}_1 \rightarrow \mathsf{state} = (\mathsf{ms}, IV_2),\\
&\mathsf{state} = (\mathsf{ms}, IV_2) \rightarrow \mathcal{C}_2 \rightarrow \mathsf{state} = (\mathsf{ms}, IV_2),\\
&\mathsf{state} = (\mathsf{ms}, IV_2) \rightarrow \mathcal{C}_3 \rightarrow \mathsf{state} = (\mathsf{ms}, \mathsf{key}_C, \mathsf{key}_S).
\end{aligned}$$

- If $\mathsf{type} =$ "open",

$$\begin{aligned}
&\mathsf{state} \rightarrow \mathcal{C}_1 \rightarrow \mathsf{state} = (\mathsf{ms}, \mathsf{key}_C, \mathsf{key}_S, IV_2),\\
&\mathsf{state} \rightarrow \mathcal{C}_2 \rightarrow \mathsf{state},
\end{aligned}$$

$$\mathsf{state} = (\mathsf{ms}, \mathsf{key}_C, \mathsf{key}_S, IV_2) \rightarrow \mathcal{C}_3 \rightarrow$$
$$\text{if it is the 1th execution, } \mathsf{state} = (\mathsf{ms}, \mathsf{key}_C, \mathsf{key}_S);$$
$$\text{if it is the 2nd execution, } \mathsf{state} = (\mathsf{key}_C, \mathsf{key}_S).$$

In the post-record phase, functionality $\mathcal{F}_{\mathsf{GP2PC}}$ updates state^* in a similar manner.

B.3 Sub-Protocol for Stateful AEAD Schemes

In Figures 15 and 16, we describe details of sub-protocol Π_{AEAD} to securely realize encryption and decryption of stateful AEAD based on AES-GCM. Protocol Π_{AEAD} works in the $(\mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{Com}})$-hybrid model, and uses $\mathcal{F}_{\mathsf{GP2PC}}$ to compute AES blocks and prove their correctness. Functionality $\mathcal{F}_{\mathsf{OLEe}}$ will be used in the generation of GMAC tags and $\mathcal{F}_{\mathsf{Com}}$ is called to prevent the possible attack of forging GMAC tags (see below for details).

To compute GMAC tags of AEAD ciphertexts, we let $\mathcal{P}$ and $\mathcal{V}$ generate additive sharings of the powers of a field element $h = \mathsf{AES}(\mathsf{key}, \mathbf{0}) \in \mathbb{F}_{2^{128}}$, following the high-level framework in DECO [ZMM+20]. While DECO computes these additive sharings using a maliciously secure 2PC protocol, we use $\mathcal{F}_{\mathsf{OLEe}}$ to generate the additive sharings of the powers of h, which is sufficient for applications (e.g., TLS) where GMAC tags are unforgeable even if at most one-bit information of h is revealed. In addition, DECO generates these additive sharings in the online phase, we first let both parties generate additive sharings $[r^i]_{2^{128}}$ for $i \in [1, m]$ in the preprocessing phase, and then transform them into additive sharings $[h^i]_{2^{128}}$ for $i \in [1, m]$ by opening $[h]_{2^{128}} - [r]_{2^{128}}$ in the online phase, where $r \in \mathbb{F}_{2^{128}}$ is a random element and m is the maximum number of AES blocks used in any AEAD ciphertext. This allows us to achieve significantly better online efficiency.

When applying sub-protocol Π_{AEAD} into our main protocol Π_{AuthData} (shown in Section 4.2), we use a label type_1 to distinguish that Π_{AEAD} is used for encryption from decryption, and a label type_2 to distinguish that a plaintext needs to kept secret from that the plaintext allows to be opened as a public value. Specifically, we have the following four cases:

- If type_1 = "encryption" and type_2 = "open", then Π_{AEAD} is used to generate a client finished message $\textsc{fin}_C$.
- If type_1 = "decryption" and type_2 = "open", Π_{AEAD} is used to generate a server finished message $\textsc{fin}_S$. In this case, an optimization is described in Section 4.3.
- If type_1 = "encryption" and type_2 = "secret", then Π_{AEAD} is used to generate AEAD ciphertexts on queries.
- If type_1 = "decryption" and type_2 = "secret", Π_{AEAD} is used to decrypt and verify AEAD ciphertexts of responses. When only one-round query-response session is executed, Π_{AEAD} is unnecessary to be invoked. When multi-round query-response sessions are executed by $\mathcal{P}$ and $\mathcal{V}$ shown in Section 4.3, both parties can execute sub-protocol Π_{AEAD} to decrypt AEAD ciphertexts from the server (see Section 4.3 for more optimization).

Note that only for the case of type_2 = "open", the additive sharings of powers of $h = \mathsf{AES}(\mathsf{key}, \mathbf{0})$ need to be generated. For the case of type_2 = "secret", these additive sharings are input to sub-protocol Π_{AEAD}.

When applying OLE with errors and additive sharings without authentication to compute GMAC tags for the case of type_1 = "decryption", a subtle issue is that a rushing adversary $\mathcal{A}$, who corrupts $\mathcal{P}$, may first get the share $\sigma_{\mathcal{V}}$ of a GMAC tag held by honest verifier $\mathcal{V}$, and then sends $\sigma'_{\mathcal{P}} = \sigma' - \sigma_{\mathcal{V}}$ to $\mathcal{V}$, where σ' is the GMAC tag involved in the AEAD ciphertext. When σ' is valid, $\mathcal{V}$ will always accept the AEAD ciphertext, even if adversary $\mathcal{A}$ adds some error into $\sigma_{\mathcal{V}}$ such that $\sigma_{\mathcal{P}} + \sigma_{\mathcal{V}} \neq \sigma'$ where $\sigma_{\mathcal{P}}$ is the share that should be obtained by $\mathcal{A}$. In this case, $\mathcal{A}$ could learn the

Protocol Π_{AEAD}

Inputs. The prover $\mathcal{P}$ and verifier $\mathcal{V}$ hold the following inputs:

- Both parties hold a state st for AEAD, the ciphertext length ℓ_{C} and header H.
- Both parties input a label $\mathsf{type}_1 \in \{\text{“encryption”}, \text{“decryption”}\}$. If $\mathsf{type}_1 = \text{“encryption”}$, $\mathcal{P}$ inputs a plaintext $\mathbf{M}$ that has been padded as $(\mathrm{M}_1, \ldots, \mathrm{M}_n)$ with $\mathrm{M}_i \in \{0,1\}^{128}$. If $\mathsf{type}_1 = \text{“decryption”}$, both parties input a ciphertext $\mathrm{CT} = (\mathbf{C}, \sigma')$ with $\mathbf{C} = (\mathrm{C}_1, \ldots, \mathrm{C}_n)$.
- The parties input a label $\mathsf{type}_2 \in \{\text{“secret”}, \text{“open”}\}$. If $\mathsf{type}_2 = \text{“open”}$, $\mathcal{V}$ inputs the same plaintext $\mathbf{M}$ for $\mathsf{type}_1 = \text{“encryption”}$, and $n = 1$ for $\mathsf{type}_1 \in \{\text{“encryption”}, \text{“decryption”}\}$. Let $m \geq n + 2$ be the maximum number of AES blocks used to generate any AEAD ciphertext in the TLS protocol execution. If $\mathsf{type}_2 = \text{“secret”}$, both parties input $\left[h^i\right]_{2^{128}}$ for all $i \in [1, m]$ where $h = \mathsf{AES}(\mathsf{key}, \mathbf{0})$.

Suppose that functionality $\mathcal{F}_{\mathsf{GP2PC}}$ stores $\mathsf{key} \in \{\mathsf{key}_C, \mathsf{key}_S\}$.

Definition of circuits. $\mathcal{P}$ and $\mathcal{V}$ define the following Boolean circuits.

- $\mathcal{C}_{aes}(\mathsf{key}, h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}}, \mathsf{st}_0, \mathsf{st}_1)$ takes as input $\mathsf{key} \in \{0,1\}^{\lambda}$, $h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}} \in \{0,1\}^{128}$ and $(\mathsf{st}_0, \mathsf{st}_1)$, and outputs $h_{\mathcal{V}} = \mathsf{AES}(\mathsf{key}, \mathbf{0}) \oplus h_{\mathcal{P}}$, $\mathrm{Z}_{0,\mathcal{V}} = \mathsf{AES}(\mathsf{key}, \mathsf{st}_0) \oplus \mathrm{Z}_{0,\mathcal{P}}$, and $\mathrm{Z}_1 = \mathsf{AES}(\mathsf{key}, \mathsf{st}_1)$. Let $\mathcal{C}'_{aes}(\mathsf{key})$ be the corresponding verification circuit that outputs $h = \mathsf{AES}(\mathsf{key}, \mathbf{0})$, $\mathrm{Z}_0 = \mathsf{AES}(\mathsf{key}, \mathsf{st}_0)$, and $\mathrm{Z}_1 = \mathsf{AES}(\mathsf{key}, \mathsf{st}_1)$ for public values $\mathsf{st}_0, \mathsf{st}_1$.
- $\mathcal{D}_{aes}(\mathsf{key}, \mathrm{Z}_{0,\mathcal{P}}, \ldots, \mathrm{Z}_{n,\mathcal{P}}, \mathsf{st}_0, \ldots, \mathsf{st}_n)$ takes as input $\mathsf{key} \in \{0,1\}^{\lambda}$, $\mathrm{Z}_{i,\mathcal{P}} \in \{0,1\}^{128}$ and st_i for each $i \in [0, n]$, and outputs $\mathrm{Z}_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}, \mathsf{st}_i) \oplus \mathrm{Z}_{i,\mathcal{P}}$ for each $i \in [0, n]$. Let $\mathcal{D}'_{aes}(\mathsf{key}, \mathrm{Z}_{1,\mathcal{P}}, \ldots, \mathrm{Z}_{n,\mathcal{P}})$ be the corresponding verification circuit, which outputs $\mathrm{Z}_0 = \mathsf{AES}(\mathsf{key}, \mathsf{st}_i)$ and $\mathrm{Z}_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}, \mathsf{st}_i) \oplus \mathrm{Z}_{i,\mathcal{P}}$ for $i \in [1, n]$, where st_i for all $i \in [0, n]$ are public values.

Preprocessing phase. Before starting a TLS session, $\mathcal{P}$ and $\mathcal{V}$ execute the following preprocessing:

1. If $\mathsf{type}_2 = \text{“open”}$, then $\mathcal{P}$ samples $h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}} \leftarrow \{0,1\}^{128}$, and $\mathcal{P}$ and $\mathcal{V}$ call functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on respective input $(\mathsf{input}, h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}})$ and (input) to let $\mathcal{F}_{\mathsf{GP2PC}}$ store $(h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}})$. If $\mathsf{type}_2 = \text{“secret”}$, then for each $i \in [0, n]$, $\mathcal{P}$ samples $\mathrm{Z}_{i,\mathcal{P}} \leftarrow \{0,1\}^{128}$, and both parties call functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on $\mathcal{P}$'s input $\mathrm{Z}_{i,\mathcal{P}}$ to have $\mathcal{F}_{\mathsf{GP2PC}}$ store $\mathrm{Z}_{i,\mathcal{P}}$. If $\mathsf{type}_2 = \text{“secret”}$, $\mathrm{Z}_{1,\mathcal{P}}, \ldots, \mathrm{Z}_{n,\mathcal{P}}$ are equivalently committed via the (input) command of $\mathcal{F}_{\mathsf{GP2PC}}$.
2. If $\mathsf{type}_2 = \text{“open”}$, $\mathcal{P}$ samples $r_{\mathcal{P}} \leftarrow \mathbb{F}_{2^{128}}$ and $\mathcal{V}$ samples $r_{\mathcal{V}} \leftarrow \mathbb{F}_{2^{128}}$, and then both parties implicitly define $r = r_{\mathcal{P}} \cdot r_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$. In this case, for $i \in [1, m]$, $\mathcal{P}$ (as a sender) and $\mathcal{V}$ (as a receiver) call functionality $\mathcal{F}_{\mathsf{OLEe}}$ on respective input $(r_{\mathcal{P}})^i$ and $(r_{\mathcal{V}})^i$ to obtain an additive sharing $\left[r^i\right]_{2^{128}} = (a_i, b_i)$ such that $a_i + b_i = (r_{\mathcal{P}})^i \cdot (r_{\mathcal{V}})^i = r^i \in \mathbb{F}_{2^{128}}$.

Handshake/record phase. When the inputs are known, $\mathcal{P}$ and $\mathcal{V}$ do the following:

3. For $i \in [0, n]$, both parties compute $\mathsf{st}_i := \mathsf{st} + i$. Relying on the value of type_2, $\mathcal{P}$ and $\mathcal{V}$ execute:
   - If $\mathsf{type}_2 = \text{“open”}$, $\mathcal{P}$ and $\mathcal{V}$ call the (eval) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on state key, Boolean circuit $\mathcal{C}_{aes}$ and $\mathcal{P}$'s input $(h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}}, \mathsf{st}_0, \mathsf{st}_1)$ to compute $h_{\mathcal{V}}$, $\mathrm{Z}_{0,\mathcal{V}}$ and Z_1. Then, both parties call the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ to let $\mathcal{V}$ obtain $(h_{\mathcal{V}}, \mathrm{Z}_{0,\mathcal{V}})$ and open Z_1 to both of them. Note that no one knows key in this phase.
   - If $\mathsf{type}_2 = \text{“secret”}$, $\mathcal{P}$ and $\mathcal{V}$ call the (eval) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on input key, circuit $\mathcal{D}_{aes}$ and $\mathcal{P}$'s input $(\mathrm{Z}_{0,\mathcal{P}}, \ldots, \mathrm{Z}_{n,\mathcal{P}}, \mathsf{st}_0, \ldots, \mathsf{st}_n)$ to compute $\mathrm{Z}_{i,\mathcal{V}}$ for $i \in [0, n]$. Then both parties call the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ to make $\mathcal{V}$ obtain $\mathrm{Z}_{i,\mathcal{V}}$ for $i \in [0, n]$.
4. Depending on the values of type_1 and type_2, $\mathcal{P}$ and $\mathcal{V}$ do the following:
   - If $\mathsf{type}_2 = \text{“open”}$, then both parties compute $\mathrm{C}_1 := \mathrm{Z}_1 \oplus \mathrm{M}_1$ and set $\mathbf{C} := \mathrm{C}_1$ if $\mathsf{type}_1 = \text{“encryption”}$, or $\mathrm{M}_1 := \mathrm{Z}_1 \oplus \mathrm{C}_1$ and set $\mathbf{M} := \mathrm{M}_1$ if $\mathsf{type}_1 = \text{“decryption”}$.
   - If $\mathsf{type}_2 = \text{“secret”}$, for each $i \in [1, n]$, $\mathcal{P}$ computes $\mathrm{B}_i := \mathrm{Z}_{i,\mathcal{P}} \oplus \mathrm{M}_i$ and sends B_i to $\mathcal{V}$ who computes $\mathrm{C}_i := \mathrm{Z}_{i,\mathcal{V}} \oplus \mathrm{B}_i$ if $\mathsf{type}_1 = \text{“encryption”}$, or $\mathcal{V}$ sends $\mathrm{Z}_{i,\mathcal{V}}$ to $\mathcal{P}$ who computes $\mathrm{M}_i := \mathrm{Z}_{i,\mathcal{P}} \oplus \mathrm{Z}_{i,\mathcal{V}} \oplus \mathrm{C}_i$ if $\mathsf{type}_1 = \text{“decryption”}$. In this case, $\mathcal{V}$ sends $\mathbf{C} := (\mathrm{C}_1, \ldots, \mathrm{C}_n)$ to $\mathcal{P}$ if $\mathsf{type}_1 = \text{“encryption”}$, or $\mathcal{P}$ sets $\mathbf{M} := (\mathrm{M}_1, \ldots, \mathrm{M}_n)$ if $\mathsf{type}_1 = \text{“decryption”}$.

Figure 15: **Protocol for securely computing AES-GCM-based stateful AEAD.**

Protocol Π_{AEAD}, continued

5. If type_2 = "open", both parties define an additive sharing $[h]_{2^{128}} = (h_{\mathcal{P}}, h_{\mathcal{V}})$. In this case, from $i = 2$ to m, $\mathcal{P}$ and $\mathcal{V}$ compute an additive sharing $\left[h^i\right]_{2^{128}}$ as follows:
   (a) Both parties run $\mathsf{Open}([h]_{2^{128}} - [r]_{2^{128}})$ to obtain $d = h - r \in \mathbb{F}_{2^{128}}$.
   (b) Let $f_i(\cdot) \in \mathbb{F}_{2^{128}}[X]$ be a polynomial such that $f_i(X) = (d + X)^i = \sum_{j=0}^{i} f_{i,j} \cdot X^j$ and thus $h^i = (d + r)^i = f_i(r)$.
   (c) Both parties compute $\left[h^i\right]_{2^{128}} := f_{i,0} + \sum_{j=1}^{i} f_{i,j} \cdot \left[r^j\right]_{2^{128}}$.

Handshake/record phase. When the inputs are known, $\mathcal{P}$ and $\mathcal{V}$ do the following:

6. $\mathcal{P}$ and $\mathcal{V}$ define an additive sharing $[\textsc{z}_0]_{2^{128}} = (\textsc{z}_{0,\mathcal{P}}, \textsc{z}_{0,\mathcal{V}})$. For $\mathbf{C} = (\textsc{c}_1, \ldots, \textsc{c}_n)$, the parties compute an additive sharing $[\sigma]_{2^{128}} = (\sigma_{\mathcal{P}}, \sigma_{\mathcal{V}})$ of tag $\sigma = \textsc{z}_0 \oplus \Phi_{(\textsc{h},\mathbf{C},\ell_{\textsc{h}},\ell_{\textsc{c}})}(h)$ as follows:
   - Let $w \in \mathbb{F}_{2^{128}}$ be the field element corresponding to bit-vector $(\ell_{\textsc{h}}, \ell_{\textsc{c}})$.
   - Both parties compute $[\sigma]_{2^{128}} := [\textsc{z}_0]_{2^{128}} + \textsc{h} \cdot \left[h^{n+2}\right]_{2^{128}} + \sum_{i=1}^{n} \textsc{c}_i \cdot \left[h^{n+2-i}\right]_{2^{128}} + w \cdot [h]_{2^{128}}$.
7. Depending on type_1, $\mathcal{P}$ and $\mathcal{V}$ output the following:
   - If type_1 = "encryption", $\mathcal{P}$ sends $\sigma_{\mathcal{P}}$ to $\mathcal{V}$, and $\mathcal{V}$ sends $\sigma_{\mathcal{V}}$ to $\mathcal{P}$ in parallel. Then both parties set $\sigma := \sigma_{\mathcal{P}} \oplus \sigma_{\mathcal{V}}$ and output $\textsc{ct} = (\mathbf{C}, \sigma)$.
   - If type_1 = "decryption", by calling functionality $\mathcal{F}_{\mathsf{Com}}$, $\mathcal{P}$ commits to $\sigma_{\mathcal{P}}$, while $\mathcal{V}$ commits to $\sigma_{\mathcal{V}}$. Then, by calling $\mathcal{F}_{\mathsf{Com}}$ again, $\mathcal{P}$ opens $\sigma_{\mathcal{P}}$ to $\mathcal{V}$, and $\mathcal{V}$ opens $\sigma_{\mathcal{V}}$ to $\mathcal{P}$. Both parties compute $\sigma := \sigma_{\mathcal{P}} \oplus \sigma_{\mathcal{V}}$, and then abort if $\sigma \neq \sigma'$ or output $\mathbf{M}$ otherwise.

Post-record phase. In this phase, after step 8 (shown below), $\mathcal{F}_{\mathsf{GP2PC}}$ stores key^* with $\mathsf{key}^* = \mathsf{key}$ in the honest case, and $\mathcal{P}$ knows key^*.

8. If type_2 = "open", both parties call the (output) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ to let $\mathcal{V}$ obtain $h_{\mathcal{P}}$ and $\textsc{z}_{0,\mathcal{P}}$, and then $\mathcal{V}$ locally computes $h := h_{\mathcal{P}} \oplus h_{\mathcal{V}}$ and $\textsc{z}_0 := \textsc{z}_{0,\mathcal{P}} \oplus \textsc{z}_{0,\mathcal{V}}$. If type_2 = "secret", both parties call the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ to let $\mathcal{V}$ receive $\textsc{z}_{0,\mathcal{P}}$, and then $\mathcal{V}$ locally computes $\textsc{z}_0 := \textsc{z}_{0,\mathcal{P}} \oplus \textsc{z}_{0,\mathcal{V}}$.
9. If type_2 = "open", both parties call the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on input key^* and circuit $\mathcal{C}'_{aes}$ to check $h = \mathsf{AES}(\mathsf{key}^*, \mathbf{0})$, $\textsc{z}_0 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}_0)$, and $\textsc{z}_1 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}_1)$. If type_2 = "secret", both parties call the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on input $(\mathsf{key}^*, \textsc{z}_{1,\mathcal{P}}, \ldots, \textsc{z}_{n,\mathcal{P}})$ and circuit $\mathcal{D}'_{aes}$ to check $\textsc{z}_0 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}_0)$ and $\textsc{z}_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}_i) \oplus \textsc{z}_{i,\mathcal{P}}$ for $i \in [1, n]$, where $(\textsc{z}_{1,\mathcal{P}}, \ldots, \textsc{z}_{n,\mathcal{P}})$ has been committed by $\mathcal{F}_{\mathsf{GP2PC}}$. In both cases, $\mathcal{V}$ aborts if the check fails.

Figure 16: **Protocol for securely computing AES-GCM-based stateful AEAD, continued.**

error without incurring abort, and then recovers $h = \mathsf{AES}(\mathsf{key}, \mathbf{0})$ from the error. To prevent this attack, we let $\mathcal{P}$ and $\mathcal{V}$ first commit to their shares of GMAC tag and then open them by calling functionality $\mathcal{F}_{\mathsf{Com}}$. This enforces adversary $\mathcal{A}$ to determine its share $\sigma'_{\mathcal{P}}$ that would be opened before seeing $\sigma_{\mathcal{V}}$, and $\mathcal{A}$ has to guess pseudorandom value h before getting key. A similar attack can be done by a malicious $\mathcal{V}$, and the countermeasure is the same. More details can be found in the following security analysis. This is not a problem for the case of type_1 = "encryption", since adversary $\mathcal{A}$ does not get a GMAC tag from the TLS server.

C Converting IT-MACs into Commitments

Our protocol will commit to queries and responses using an additively homomorphic commitment (AHC) scheme, which is modeled in functionality $\mathcal{F}_{\mathsf{HCom}}$ shown in Figure 10. In the main protocol Π_{AuthData} (shown in Section 4.2), we use a conversion functionality $\mathcal{F}_{\mathsf{Conv}}$ (shown in Figure 17) to convert IT-MACs into AHCs. The definition of the ($\mathsf{convert}$) command of functionality $\mathcal{F}_{\mathsf{Conv}}$ is

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

This functionality interacts with $\mathcal{P}$ and $\mathcal{V}$, and has all the features of $\mathcal{F}_{\mathsf{IZK}}$ (shown in Figure 11) and $\mathcal{F}_{\mathsf{HCom}}$ (shown in Figure 10). Furthermore, this functionality involves the following commands.

Conversion. Upon receiving $(\mathsf{convert}, \mathsf{id}, \mathsf{cid})$ from $\mathcal{P}$ and $\mathcal{V}$, if $(\mathsf{id}, [\![x]\!])$ was previously stored, then store (cid, x).

Reveal global key. Upon receiving $(\mathsf{revealkey})$ from $\mathcal{P}$ and $\mathcal{V}$, if Δ was previously stored, then send Δ to $\mathcal{P}$. Ignore all commands associated with Δ.

Figure 17: **Functionality for converting IT-MACs to additively homomorphic commitments.**

similar to that of the ideal functionalities in prior works, e.g., [EGK+20, WYX+21]. Functionality $\mathcal{F}_{\mathsf{Conv}}$ additionally includes a $(\mathsf{revealkey})$ command which reveals global key Δ to $\mathcal{P}$ and hereafter ignores all commands related to Δ. In this section, we present a protocol to securely realize functionality $\mathcal{F}_{\mathsf{Conv}}$ with $\mathsf{revealkey}$, and show how to simply extend this protocol to realize $\mathcal{F}_{\mathsf{Conv}}$ without $\mathsf{revealkey}$.

In Figure 18, we show an efficient protocol Π_{Conv} to securely instantiate functionality $\mathcal{F}_{\mathsf{Conv}}$. This protocol enables two parties to convert authenticated bits into AHCs with message space of a large field $\mathbb{F}$. For example, we can convert such IT-MACs into Pedersen commitments [Ped92] or KZG polynomial commitments [KZG10] (by packing multiple values together). Then, the Pedersen or KZG commitments can be used in zk-SNARKs such as [CFQ19, GWC19, MBKM19, CHM+20, CFF+21] to accelerate generation of the proofs on statements w.r.t. queries and responses. We focus on the case that $\mathbb{F}$ is a finite field modulo a large prime q (i.e., $\mathbb{F} = \mathbb{Z}_q$), but our protocol supports any large field $\mathbb{F}$. [5]

We realize conversion in two steps: (1) convert IT-MACs over a binary field $\mathbb{F}_{2^\lambda}$ into that over a prime field $\mathbb{F}$ using a random oracle [IKNP03, CKKZ12, GKWY20, GKMN21]; (2) convert IT-MACs over $\mathbb{F}$ into AHCs using a random linear combination. In protocol Π_{Conv} (Figure 18), we use functionality $\mathcal{F}_{\mathsf{HCom}}$ to commit to the bits of queries and responses (i.e., $\boldsymbol{u}$) rather than using an AHC scheme. This allows us to simplify the proof of security. When instantiating $\mathcal{F}_{\mathsf{HCom}}$ with an AHC scheme, $\mathcal{P}$ can output the randomness that is used to generate these AHCs, and in turn the randomness will be used by $\mathcal{P}$ as a part of witness in subsequent ZK proofs on these AHCs.

In step 3 of protocol Π_{Conv}, $\mathcal{V}$ sends field elements $\{W_i\}$, that encrypt global key Γ, to $\mathcal{P}$. A malicious party $\mathcal{V}$ may introduce some errors into these field elements, which allows it to learn some bits of secret vector $\boldsymbol{u}$. To prevent such attack, we adopt the commit-then-open approach: $\mathcal{V}$ first commits to global keys Δ, Γ, and then opens them to $\mathcal{P}$ who can check the correctness of $\{W_i\}$ with Δ, Γ and its MAC tags. In this case, we also let $\mathcal{P}$ commit to the MAC tag $\widetilde{\mathsf{M}}[y]$ that will be opened later, before Δ, Γ are opened. This assures that $\mathcal{P}$ cannot forge a MAC tag on an inconsistent value y after knowing Δ, Γ.

In the case that $\mathcal{V}$ is malicious, it may commit to an inconsistent global key $\Delta' = \Delta \oplus E$ for an error E chosen by $\mathcal{V}$. Then, $\mathcal{V}$ could open Δ' along with Γ to honest prover $\mathcal{P}$ who uses $\mathsf{K}'[u_i] = \mathsf{M}[u_i] \oplus u_i\Delta \oplus u_i E$ to check the correctness of W_i for $i \in [1, n]$. For $i \in [1, n]$, malicious verifier $\mathcal{V}$ can construct $W_i = \mathsf{H}(\mathsf{K}[u_i]) - \mathsf{H}(\mathsf{K}[u_i] \oplus \Delta) + \Gamma$ to guess $u_i = 0$ or $W_i = \mathsf{H}(\mathsf{K}[u_i] \oplus E) - \mathsf{H}(\mathsf{K}[u_i] \oplus \Delta \oplus E) + \Gamma$ to guess $u_i = 1$, where $\mathsf{K}[u_i] = \mathsf{M}[u_i] \oplus u_i\Delta$. This allows $\mathcal{V}$ to perform a selective failure attack on some bits of $\boldsymbol{u}$ by observing if $\mathcal{P}$ aborts. We prevent such attack by letting two parties additionally input $[\![\boldsymbol{v}]\!]$ with a random $\boldsymbol{v} \in \{0,1\}^\lambda$, where $[\![\boldsymbol{v}]\!]$ can be efficiently generated by calling the $(\mathsf{authinput})$ command of functionality $\mathcal{F}_{\mathsf{IZK}}$ or running the COT protocol with the same global key Δ. When Δ is revealed, for each $i \in [1, \lambda]$, $\mathcal{V}$ needs to send $\mathsf{K}[v_i]$ to $\mathcal{P}$ who checks $\mathsf{K}[v_i] = \mathsf{M}[v_i] \oplus v_i\Delta$. If

[5] In this paper, we say that $\mathbb{F}$ is a large field if $|\mathbb{F}| \geq 2^\lambda$.

Protocol Π_{Conv}

Inputs. Parties $\mathcal{P}$ and $\mathcal{V}$ hold the following inputs:

- $\mathcal{V}$ holds a uniform global key $\Delta \in \{0,1\}^\lambda$. Both parties input a vector of IT-MACs $[\![\boldsymbol{u}]\!] = (\boldsymbol{u}, \mathsf{M}[\boldsymbol{u}], \mathsf{K}[\boldsymbol{u}])$ with $\boldsymbol{u} = (u_1, \ldots, u_n) \in \{0,1\}^n$ and $\mathsf{M}[u_i] = \mathsf{K}[u_i] \oplus u_i\Delta$ for $i \in [1, n]$.
- Both parties input $[\![\boldsymbol{v}]\!] = (\boldsymbol{v}, \mathsf{M}[\boldsymbol{v}], \mathsf{K}[\boldsymbol{v}])$ with a random vector $\boldsymbol{v} = (v_1, \ldots, v_\lambda) \in \{0,1\}^\lambda$ and $\mathsf{M}[v_i] = \mathsf{K}[v_i] \oplus v_i\Delta$ for $i \in [1, \lambda]$.
- Let $\mathsf{H} : \{0,1\}^\lambda \to \mathbb{F}$ be a random oracle, and $\mathsf{H}' : \{0,1\}^* \to \{0,1\}^\lambda$ be another random oracle.

Preprocessing phase. $\mathcal{P}$ and $\mathcal{V}$ execute the following preprocessing:

1. $\mathcal{V}$ samples $\Gamma \leftarrow \mathbb{F}$, and then commits to (Δ, Γ) by calling functionality $\mathcal{F}_{\mathsf{Com}}$.
2. $\mathcal{P}$ samples $u_0 \leftarrow \mathbb{F}$. Then, $\mathcal{P}$ (acting as a sender) and $\mathcal{V}$ (acting as a receiver) call $\mathcal{F}_{\mathsf{OLEe}}$ on respective input u_0 and Γ. Functionality $\mathcal{F}_{\mathsf{OLEe}}$ sends $\widetilde{\mathsf{M}}[u_0] \in \mathbb{F}$ to $\mathcal{P}$ and $-\widetilde{\mathsf{K}}[u_0] \in \mathbb{F}$ to $\mathcal{V}$ such that $\widetilde{\mathsf{M}}[u_0] = \widetilde{\mathsf{K}}[u_0] + u_0 \cdot \Gamma \in \mathbb{F}$. Let $[\![u_0]\!]_{\mathbb{F}} = (u_0, \widetilde{\mathsf{M}}[u_0], \widetilde{\mathsf{K}}[u_0])$.

Online phase. $\mathcal{P}$ and $\mathcal{V}$ convert $[\![\boldsymbol{u}]\!]$ into additive homomorphic commitments of $\boldsymbol{u}$ over a large field $\mathbb{F}$.

3. For $i \in [1, n]$, $\mathcal{P}$ and $\mathcal{V}$ convert $[\![u_i]\!]$ into $[\![u_i]\!]_{\mathbb{F}}$ as follows:
   (a) $\mathcal{V}$ computes $W_i := \mathsf{H}(\mathsf{K}[u_i]) - \mathsf{H}(\mathsf{K}[u_i] \oplus \Delta) + \Gamma \in \mathbb{F}$ and sets $\widetilde{\mathsf{K}}[u_i] := \mathsf{H}(\mathsf{K}[u_i]) \in \mathbb{F}$, and then sends W_i to $\mathcal{P}$.
   (b) $\mathcal{P}$ computes $\widetilde{\mathsf{M}}[u_i] := \mathsf{H}(\mathsf{M}[u_i]) + u_i \cdot W_i \in \mathbb{F}$, where $\widetilde{\mathsf{M}}[u_i] = \widetilde{\mathsf{K}}[u_i] + u_i \cdot \Gamma$.
   (c) Both parties define $[\![u_i]\!]_{\mathbb{F}} = (u_i, \widetilde{\mathsf{M}}[u_i], \widetilde{\mathsf{K}}[u_i])$.
4. For $i \in [0, n]$, $\mathcal{P}$ commits to u_i by sending $(\mathsf{commit}, \mathsf{cid}_i, u_i)$ to functionality $\mathcal{F}_{\mathsf{HCom}}[\mathbb{F}]$ which sends $(\mathsf{committed}, \mathsf{cid}_i)$ to $\mathcal{V}$.
5. $\mathcal{V}$ samples $\chi_1, \ldots, \chi_n \leftarrow \mathbb{F}$ and sends them to $\mathcal{P}$. Both parties *locally* compute $[\![y]\!]_{\mathbb{F}} := \sum_{i\in[1,n]} \chi_i \cdot [\![u_i]\!]_{\mathbb{F}} + [\![u_0]\!]_{\mathbb{F}}$. $\mathcal{P}$ and $\mathcal{V}$ compute a linear combination of the commitments of $u_0, u_1, \ldots, u_n$ by sending $(\mathsf{lincomb}, \mathsf{cid}', \mathsf{cid}_0, \ldots, \mathsf{cid}_n, \chi_1, \ldots, \chi_n)$ to functionality $\mathcal{F}_{\mathsf{HCom}}[\mathbb{F}]$ which stores (cid', y) with $y = \sum_{i\in[1,n]} \chi_i \cdot u_i + u_0$.
6. $\mathcal{P}$ commits to $\widetilde{\mathsf{M}}[y] \in \mathbb{F}$ by calling functionality $\mathcal{F}_{\mathsf{Com}}$. Then, $\mathcal{V}$ opens (Δ, Γ) to $\mathcal{P}$ by calling functionality $\mathcal{F}_{\mathsf{Com}}$. In parallel, $\mathcal{V}$ computes $\tau := \mathsf{H}'(\mathsf{K}[v_1], \ldots, \mathsf{K}[v_\lambda])$, and sends $(\tau, \widetilde{\mathsf{K}}[y])$ to $\mathcal{P}$.
7. For $i \in [1, n]$, $\mathcal{P}$ computes $\mathsf{K}[u_i] := \mathsf{M}[u_i] \oplus u_i\Delta$ and checks $W_i = \mathsf{H}(\mathsf{K}[u_i]) - \mathsf{H}(\mathsf{K}[u_i] \oplus \Delta) + \Gamma$. Then, $\mathcal{P}$ computes $\tau' := \mathsf{H}'(\mathsf{M}[v_1] \oplus v_1\Delta, \ldots, \mathsf{M}[v_\lambda] \oplus v_\lambda\Delta)$ and checks $\tau' = \tau$. $\mathcal{P}$ also checks $\widetilde{\mathsf{M}}[y] = \widetilde{\mathsf{K}}[y] + y \cdot \Gamma$. If any check fails, $\mathcal{P}$ aborts.
8. $\mathcal{P}$ opens the commitment of y by sending $(\mathsf{open}, \mathsf{cid}')$ to functionality $\mathcal{F}_{\mathsf{HCom}}[\mathbb{F}]$ which sends $(\mathsf{opened}, \mathsf{cid}', y)$ to $\mathcal{V}$. In parallel, $\mathcal{P}$ opens $\widetilde{\mathsf{M}}[y]$ to $\mathcal{V}$ by calling $\mathcal{F}_{\mathsf{Com}}$. $\mathcal{V}$ checks that $\widetilde{\mathsf{M}}[y] = \widetilde{\mathsf{K}}[y] + y \cdot \Gamma$ and aborts if the check fails.
9. For $i \in [1, n]$, both parties output $(\mathsf{cid}_1, \ldots, \mathsf{cid}_n)$ that represents the identifiers of commitments of $u_1, \ldots, u_n$.

Figure 18: **Protocol for converting IT-MACs to AHCs in the $(\mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{HCom}}, \mathcal{F}_{\mathsf{OLEe}})$-hybrid model.**

$\mathcal{V}$ opens an inconsistent Δ', it has to guess $\boldsymbol{v} \in \{0,1\}^\lambda$ correctly, which occurs with probability at most $1/2^\lambda$. We can use a random oracle H' to compress communication of sending $\mathsf{K}[\boldsymbol{v}]$ from λ^2 bits to λ bits. Similarly, a malicious $\mathcal{V}$ may use an inconsistent Γ' when calling functionality $\mathcal{F}_{\mathsf{OLEe}}$, which allows it to reveal u_0 and thus the linear combination $\sum_{i\in[1,n]} \chi_i \cdot u_i$. We prevent the attack by letting $\mathcal{V}$ send $\widetilde{\mathsf{K}}[y]$ to $\mathcal{P}$ who checks $\widetilde{\mathsf{M}}[y] = \widetilde{\mathsf{K}}[y] + y \cdot \Gamma$ before y is opened.

We can extend protocol Π_{Conv} to securely realize functionality $\mathcal{F}_{\mathsf{Conv}}$ without the $(\mathsf{revealkey})$ command. Specifically, two parties $\mathcal{P}$ and $\mathcal{V}$ converts $[\![\boldsymbol{u}]\!]_\Delta$ into $[\![\boldsymbol{u}]\!]_{\Delta'}$ for an independent random global key Δ', and then use $[\![\boldsymbol{u}]\!]_{\Delta'}$ to execute the protocol. In this way, only Δ' is revealed, and

Δ is still kept secret. Thus, $[\![\boldsymbol{u}]\!]_\Delta$ can still be used in other protocol executions. The remaining task is to generate $[\![\boldsymbol{u}]\!]_{\Delta'}$ with a consistent vector $\boldsymbol{u}$. $\mathcal{P}$ and $\mathcal{V}$ can produce $[\![\boldsymbol{u}]\!]_{\Delta'}$ by executing a COT protocol. However, a malicious $\mathcal{P}$ may adopt an inconsistent vector $\boldsymbol{u}'$ in the COT protocol execution. This can be detected by checking the consistency of $[\![\boldsymbol{u}]\!]_\Delta$ and $[\![\boldsymbol{u}]\!]_{\Delta'}$ using a random linear combination. In particular, $\mathcal{V}$ samples $\psi_1, \ldots, \psi_n \leftarrow \mathbb{F}_{2^\lambda}$ and sends them to $\mathcal{P}$. Then, both parties locally compute

$$[\![z]\!]_\Delta := \sum_{i \in [1,n]} \psi_i \cdot [\![u_i]\!]_\Delta + [\![r]\!]_\Delta, \; [\![z]\!]_{\Delta'} := \sum_{i \in [1,n]} \psi_i \cdot [\![u_i]\!]_{\Delta'} + [\![r]\!]_{\Delta'},$$

where $[\![r]\!]_\Delta$ and $[\![r]\!]_{\Delta'}$ are IT-MACs on a random value $r \in \mathbb{F}_{2^\lambda}$, and they can be generated by running the COT protocol. Here r is used to mask $\sum_{i \in [1,n]} \psi_i \cdot u_i \in \mathbb{F}_{2^\lambda}$. A malicious $\mathcal{P}$ may incur the inconsistency of r in $[\![r]\!]_\Delta$ and $[\![r]\!]_{\Delta'}$, which would has no impact on security. $\mathcal{P}$ can send z to $\mathcal{V}$, who checks that both $[\![z]\!]_\Delta - z$ and $[\![z]\!]_{\Delta'} - z$ are IT-MACs on zero. If the vector $\boldsymbol{u}$ is inconsistent in $[\![\boldsymbol{u}]\!]_\Delta$ and $[\![\boldsymbol{u}]\!]_{\Delta'}$, this check passes with negligible probability, as ψ_i for all $i \in [1, n]$ are sampled at random after $[\![\boldsymbol{u}]\!]_\Delta$ and $[\![\boldsymbol{u}]\!]_{\Delta'}$ have been defined. Combining with the above approach, our approach underlying the protocol Π_{Conv} is easy to be extended to convert additively homomorphic commitments into IT-MACs.

Theorem 2. *Protocol Π_{Conv} (shown in Figure 18) securely realizes functionality $\mathcal{F}_{\mathsf{Conv}}$ (shown in Figure 17) in the ($\mathcal{F}_{\mathsf{Com}}$, $\mathcal{F}_{\mathsf{HCom}}, \mathcal{F}_{\mathsf{OLEe}}$)-hybrid model, if H is a random oracle.*

Proof. We first consider the case of a malicious prover $\mathcal{P}$ and then consider the case of a malicious verifier $\mathcal{V}$. In each case, we construct a probabilistic polynomial time (PPT) simulator $\mathcal{S}$ given access to functionality $\mathcal{F}_{\mathsf{Conv}}$, which runs a PPT adversary $\mathcal{A}$ as a subroutine, and emulates functionalities $\mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{HCom}}, \mathcal{F}_{\mathsf{OLEe}}$. Whenever $\mathcal{A}$ or the honest party simulated by $\mathcal{S}$ will abort, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{Conv}}$, and then aborts. In both cases, $\mathcal{S}$ simulates random oracle H' by answering a random string in $\{0,1\}^\lambda$ for each query while keeping the consistency of answers.

Malicious prover $\mathcal{P}$. In this case, $\mathcal{S}$ knows $(u_i, \mathsf{M}[u_i])$ for $i \in [1, n]$ and $(v_i, \mathsf{M}[v_i])$ for $i \in [1, \lambda]$. $\mathcal{S}$ samples $\Gamma \leftarrow \mathbb{F}$. For each query Y of random oracle H, $\mathcal{S}$ responds as follows:

- Before (Δ, Γ) is revealed, if Y was previously queried, retrieve (Y, Z) and send Z to $\mathcal{A}$. Otherwise, sample $Z \leftarrow \mathbb{F}$, store (Y, Z) and send Z to $\mathcal{A}$.
- After (Δ, Γ) is revealed, respond as above, except for the following difference: if $Y = \mathsf{M}[u_i] \oplus \Delta$ for some $i \in [1, n]$, retrieve $(\mathsf{M}[u_i], Z_i)$, set $Z := Z_i - W_i + \Gamma$ if $u_i = 0$ or $Z := W_i + Z_i - \Gamma$ otherwise, then store (Y, Z) and send Z to $\mathcal{A}$.

By emulating functionalities $\mathcal{F}_{\mathsf{Com}}$, $\mathcal{F}_{\mathsf{HCom}}$ and $\mathcal{F}_{\mathsf{OLEe}}$, $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows.

1. In the preprocessing phase, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OLEe}}$ by receiving u_0, $\widetilde{\mathsf{M}}[u_0]$ and an error vector $\boldsymbol{e}$ sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{OLEe}}$.
2. For each $i \in [1, n]$, $\mathcal{S}$ samples $W_i \leftarrow \mathbb{F}$ and sends it to $\mathcal{A}$. Then $\mathcal{S}$ computes $\widetilde{\mathsf{M}}[u_i]$.
3. $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{HCom}}[\mathbb{F}]$ by receiving u_i' from $\mathcal{A}$ for $i \in [0, n]$. Then, $\mathcal{S}$ computes $e_i := u_i' - u_i \in \mathbb{F}$ for $i \in [0, n]$.
4. $\mathcal{S}$ samples $\chi_1, \ldots, \chi_n \leftarrow \mathbb{F}$ and sends them to adversary $\mathcal{A}$. Then, $\mathcal{S}$ computes $\widetilde{\mathsf{M}}[y]$ with $(\widetilde{\mathsf{M}}[u_0], \widetilde{\mathsf{M}}[u_1], \ldots, \widetilde{\mathsf{M}}[u_n])$.
5. $\mathcal{S}$ emulates the (commit) command of $\mathcal{F}_{\mathsf{Com}}$ by receiving $\widetilde{\mathsf{M}}[y]'$ from $\mathcal{A}$. Then, $\mathcal{S}$ computes $E := \widetilde{\mathsf{M}}[y]' - \widetilde{\mathsf{M}}[y] \in \mathbb{F}$. Next, $\mathcal{S}$ receives Δ from functionality $\mathcal{F}_{\mathsf{Conv}}$. $\mathcal{S}$ emulates the (open) command of $\mathcal{F}_{\mathsf{Com}}$ by sending (Δ, Γ) to $\mathcal{A}$.

6. $\mathcal{S}$ computes $\mathsf{K}[v_i] := \mathsf{M}[v_i] \oplus v_i\Delta$ for $i \in [1,\lambda]$ and sets $\tau := \mathsf{H}(\mathsf{K}[v_1], \ldots, \mathsf{K}[v_\lambda])$. $\mathcal{S}$ also computes $\widetilde{\mathsf{K}}[u_i] := \widetilde{\mathsf{M}}[u_i] - u_i \cdot \Gamma \in \mathbb{F}$ for $i \in [1,n]$ and $\widetilde{\mathsf{K}}[u_0] := \widetilde{\mathsf{M}}[u_0] - u_0 \cdot \Gamma - (\boldsymbol{g} * \boldsymbol{e}) \odot \boldsymbol{\Gamma} \in \mathbb{F}$, then computes $\widetilde{\mathsf{K}}[y] := \sum_{i\in[1,n]} \chi_i \cdot \widetilde{\mathsf{K}}[u_i] + \widetilde{\mathsf{K}}[u_0] \in \mathbb{F}$. Then, $\mathcal{S}$ sends $(\tau, \widetilde{\mathsf{K}}[y])$ to $\mathcal{A}$.
7. $\mathcal{S}$ checks that $E = (\sum_{i\in[1,n]} \chi_i \cdot e_i + e_0) \cdot \Gamma + (\boldsymbol{g} * \boldsymbol{e}) \odot \boldsymbol{\Gamma}$ and $e_i = 0$ for all $i \in [1,n]$, where vector $\boldsymbol{\Gamma}$ is the bit-decomposition of $\Gamma \in \mathbb{F}$. If the check fails, $\mathcal{S}$ aborts.

It is easy to see that the simulation of functionalities $\mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{HCom}}, \mathcal{F}_{\mathsf{OLEe}}$ is perfect. After obtaining Δ, $\mathcal{S}$ is able to compute $\mathsf{K}[v_i]$ for $i \in [1,\lambda]$, and then uses them to simulate τ perfectly. Similarly, after knowing Γ, $\mathcal{S}$ can compute $\widetilde{\mathsf{K}}[u_i]$ for $i \in [0,n]$, and then uses them to compute $\widetilde{\mathsf{K}}[y]$ that has the same distribution as the one sent in the real protocol execution. The simulation of random oracle H is also perfect, unless $\mathcal{A}$ makes a query $\mathsf{M}[u_i] \oplus \Delta$ for some $i \in [1,n]$ to H before Δ is revealed. The bad event happens with negligible probability, since $\Delta \in \{0,1\}^\lambda$ is unknown for $\mathcal{A}$. Before Δ is revealed, W_i for $i \in [1,n]$ in the real protocol execution are uniform. This because $\Delta \in \{0,1\}^\lambda$ is uniform and H is a random oracle, and thus the probability that $\mathcal{A}$ makes a query $\mathsf{M}[u_i] \oplus \Delta$ to random oracle H for some $i \in [1,n]$ is negligible in λ. The elements $W_1, \ldots, W_n$ in the real protocol are computationally indistinguishable from that simulated by $\mathcal{S}$. For a malicious $\mathcal{P}$, checking $\widetilde{\mathsf{M}}[y] = \widetilde{\mathsf{K}}[y] + y \cdot \Gamma$ in the real protocol execution is equivalent to check $E = (\sum_{i\in[1,n]} \chi_i \cdot e_i + e_0) \cdot \Gamma + (\boldsymbol{g} * \boldsymbol{e}) \odot \boldsymbol{\Gamma}$ in the ideal-world execution. The only difference is that $\mathcal{S}$ additionally checks $e_i = 0$ for all $i \in [1,n]$.

Below, we show that $e_i = 0$ for all $i \in [1,n]$, if honest $\mathcal{V}$ does not abort in the real protocol execution. As analyzed above, W_i for each $i \in [1,n]$ is computationally indistinguishable from a uniform element in the real protocol execution. Therefore, Γ is computationally indistinguishable from a uniform element in $\mathbb{F}$ before it is revealed. Thus, we have $\sum_{i\in[1,n]} \chi_i \cdot e_i + e_0 = 0$. The coefficients $\chi_1, \ldots, \chi_n \in \mathbb{F}$ are sampled uniformly after the errors $e_0, e_1, \ldots, e_n$ have been defined. Therefore, $e_i = 0$ for all $i \in [1,n]$, except with probability $1/|\mathbb{F}|$ that is negligible. Overall, the checking performed by $\mathcal{S}$ is computationally indistinguishable from that performed by $\mathcal{V}$ in the real protocol execution. In addition, we also obtain that $(\boldsymbol{g} * \boldsymbol{e}) \odot \boldsymbol{\Gamma} = 0$, which allows $\mathcal{A}$ to reveal one-bit information of Γ on average. This is harmless as $\Gamma \in \mathbb{F}$ still has a sufficiently high entropy before it is revealed and $\mathcal{A}$ would always obtain Γ in step 6.

In conclusion, the joint distribution of the outputs of $\mathcal{A}$ and $\mathcal{V}$ in the real-world execution is computationally indistinguishable from that of $\mathcal{S}$ and $\mathcal{V}$ in the ideal-world execution.

Malicious verifier $\mathcal{V}$. In this case, $\mathcal{S}$ knows Δ, $\mathsf{K}[u_i]$ for $i \in [1,n]$ and $\mathsf{K}[v_i]$ for $i \in [1,\lambda]$. $\mathcal{S}$ simulates a random oracle H by answering a random field element in $\mathbb{F}$ for each query while keeping the consistency of answers. $\mathcal{S}$ emulates functionalities $\mathcal{F}_{\mathsf{Com}}$, $\mathcal{F}_{\mathsf{HCom}}$ and $\mathcal{F}_{\mathsf{OLEe}}$, and interacts with $\mathcal{A}$ as follows.

1. In the preprocessing phase, $\mathcal{S}$ emulates the (commit) command of functionality $\mathcal{F}_{\mathsf{Com}}$ by receiving (Δ', Γ) from $\mathcal{A}$.
2. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OLEe}}$ by receiving Γ' and $-\widetilde{\mathsf{K}}[u_0]$ from $\mathcal{A}$.
3. For each $i \in [1,n]$, after receiving W_i from $\mathcal{A}$, simulator $\mathcal{S}$ sets $\widetilde{\mathsf{K}}[u_i] = \mathsf{H}(\mathsf{K}[u_i])$.
4. For AHCs of $u_0, u_1, \ldots, u_n$, $\mathcal{S}$ emulates the (commit) command of $\mathcal{F}_{\mathsf{HCom}}[\mathbb{F}]$ by sending cid_i for $i \in [0,n]$ to $\mathcal{A}$.
5. After receiving $\chi_1, \ldots, \chi_n$ from $\mathcal{A}$, $\mathcal{S}$ computes $\widetilde{\mathsf{K}}[y]$ following the protocol description.
6. $\mathcal{S}$ receives $(\tau, \widetilde{\mathsf{K}}'[y])$ from $\mathcal{A}$. Then $\mathcal{S}$ checks that $\tau = \mathsf{H}'(\mathsf{K}[v_1], \ldots, \mathsf{K}[v_\lambda])$ and $\Delta' = \Delta$. $\mathcal{S}$ also checks $\widetilde{\mathsf{K}}'[y] = \widetilde{\mathsf{K}}[y]$ and $\Gamma' = \Gamma$. If any check fails, $\mathcal{S}$ aborts.
7. $\mathcal{S}$ uses $\mathsf{K}[u_i]$ for $i \in [1,n]$ and (Δ, Γ) to check correctness of W_i for all $i \in [1,n]$ following the protocol description.

8. $\mathcal{S}$ emulates the (open) command of functionality $\mathcal{F}_{\mathsf{HCom}}[\mathbb{F}]$ by sending a random element $y \in \mathbb{F}$ to $\mathcal{A}$. Then, $\mathcal{S}$ computes $\widetilde{\mathsf{M}}[y] = \widetilde{\mathsf{K}}[y] + y \cdot \Gamma$, and emulates the (open) command of $\mathcal{F}_{\mathsf{Com}}$ by sending $\widetilde{\mathsf{M}}[y]$ to $\mathcal{A}$.

It is clear that the simulation of functionalities $\mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{HCom}}, \mathcal{F}_{\mathsf{OLEe}}$ is perfect. In the following, we show that all checks performed by honest prover $\mathcal{P}$ in the real protocol execution are statistically indistinguishable from that performed by $\mathcal{S}$ in the ideal-world execution. Let $E_1 = \Delta \oplus \Delta'$, $E_2 = \Gamma' - \Gamma$ and $E_3 = \widetilde{\mathsf{K}}'[y] - \widetilde{\mathsf{K}}[y]$. If $E_1 \neq 0$ and $\mathcal{P}$ does not abort, then $\tau = \mathsf{H}'(\mathsf{M}[v_1] \oplus v_1\Delta \oplus v_1E_1, \ldots, \mathsf{M}[v_\lambda] \oplus v_\lambda\Delta \oplus v_\lambda E_1) = \mathsf{H}'(\mathsf{K}[v_1] \oplus v_1E_1, \ldots, \mathsf{K}[v_\lambda] \oplus v_\lambda E_1)$. If $\mathcal{A}$ makes a query $(\mathsf{K}[v_1] \oplus v_1E_1, \ldots, \mathsf{K}[v_\lambda] \oplus v_\lambda E_1)$ to random oracle H', then $\mathcal{A}$ succeeds to guess $\boldsymbol{v}$ which occurs with probability $1/2^\lambda$ as $\boldsymbol{v} \in \{0,1\}^\lambda$ is uniform. Otherwise, $\mathcal{A}$ guesses τ correctly, which happens with probability $1/2^\lambda$, as $\tau = \mathsf{H}'(\mathsf{K}[v_1] \oplus v_1E_1, \ldots, \mathsf{K}[v_\lambda] \oplus v_\lambda E_1)$ is uniformly random in the random-oracle model. Therefore, checking τ and $\Delta' = \Delta$ in the ideal-world execution is statistically indistinguishable from checking τ in the real protocol execution. Additionally, due to $E_1 = 0$, $\mathcal{P}$ always computes $\mathsf{K}[u_i] = \mathsf{M}[u_i] \oplus u_i\Delta$, and thus the check of W_i for all $i \in [1, n]$ is identical in both worlds. Then we obtain that $\widetilde{\mathsf{M}}[u_i] = \widetilde{\mathsf{K}}[u_i] + u_i \cdot \Gamma$. If $E_2 \neq 0$, then $\widetilde{\mathsf{M}}[u_0] = \widetilde{\mathsf{K}}[u_0] + u_0 \cdot \Gamma + u_0 \cdot E_2$. Thus, $\widetilde{\mathsf{M}}[y] = \widetilde{\mathsf{K}}[y] + y \cdot \Gamma + u_0 \cdot E_2$. If $\mathcal{P}$ does not abort in the real protocol execution, then $\widetilde{\mathsf{K}}'[y] = \widetilde{\mathsf{M}}[y] - y \cdot \Gamma$, meaning that $E_3 = u_0 \cdot E_2$. Since E_2, E_3 have been defined before y is opened, we have $E_2 = E_3 = 0$ based on the fact that $u_0 \in \mathbb{F}$ is uniform before y is opened. Therefore, checking $\widetilde{\mathsf{K}}'[y]$ and $\Gamma' = \Gamma$ in the ideal-world execution is statistically indistinguishable from checking $\widetilde{\mathsf{K}}'[y]$ in the real protocol execution. From the fact that $u_0 \in \mathbb{F}$ is uniform, $y \in \mathbb{F}$ opened by $\mathcal{P}$ in the real protocol execution is uniformly random in $\mathbb{F}$. This means that y simulated by $\mathcal{S}$ has the identical distribution as that in the real protocol execution. Due to $E_2 = 0$, we have that $\widetilde{\mathsf{M}}[y] = \widetilde{\mathsf{K}}[y] + y \cdot \Gamma$. Hence, $\widetilde{\mathsf{M}}[y]$ opened by $\mathcal{S}$ also has the identical distribution as that in the real protocol execution.

In conclusion, the joint distribution of the outputs of $\mathcal{A}$ and $\mathcal{P}$ in the real-world execution is statistically indistinguishable from that of $\mathcal{S}$ and $\mathcal{P}$ in the ideal-world execution. □

Optimizations. We can further optimize the efficiency of protocol Π_{Conv} as follows:

- When instantiating $\mathcal{F}_{\mathsf{HCom}}$ with an additively homomorphic commitment scheme with algorithm Commit, we are able to use the Fiat-Shamir transformation to generate random challenges $\chi_1, \ldots, \chi_n$ without any interaction. In particular, $\mathcal{P}$ and $\mathcal{V}$ can compute $(\chi_1, \ldots, \chi_n) := \mathsf{H}''(W_1, \ldots, W_n, \mathsf{Commit}(u_1), \ldots, \mathsf{Commit}(u_n))$, where H'' is a random oracle and $\mathsf{Commit}(u_i)$ is an AHC on bit u_i for $i \in [1, n]$. Note that $|\mathbb{F}| \geq 2^\lambda$ and $\mathbb{F}$ is sufficiently large to support Fiat-Shamir. The challenges $\psi_1, \ldots, \psi_n$ used for converting IT-MACs between different global keys can be computed in a similar way based on the Fiat-Shamir transformation.
- In protocol Π_{Conv} (Figure 18), we let $\mathcal{P}$ commit to every bit u_i, which requires to generate n AHCs and IT-MACs over $\mathbb{F}$, where n is the length of $\boldsymbol{u}$. We can reduce the communication and computational costs of generating these AHCs from $O(n)$ to $O(n/m)$ by first packing IT-MACs of m bits together and then converting them into AHCs. Here m is a parameter depending on the applications, and satisfies that $2^m - 1 < |\mathbb{F}|$ in order to avoid overflow. Recall that we focus on the case of $\mathbb{F} = \mathbb{Z}_q$ for a prime q. Then, after $\mathcal{P}$ and $\mathcal{V}$ generated $[\![u_i]\!]_{\mathbb{F}}$ for $i \in [1, n]$, both parties can compute $[\![v_j]\!]_{\mathbb{F}} := \sum_{i=1}^{m} 2^{i-1} \cdot [\![u_{(j-1)m+i}]\!]_{\mathbb{F}}$ for $j \in [1, \ell]$ where $\ell = \lceil n/m \rceil$ and v_j is an integer of m bits. Using the same approach, both parties can convert $[\![v_j]\!]_{\mathbb{F}}$ into $\mathsf{Commit}(v_j)$ for $j \in [1, \ell]$, where Commit denotes the commit algorithm of an AHC scheme.

Protocol Π_{GP2PC}

Inputs. $\mathcal{P}$ (acting as the garbler and prover) and $\mathcal{V}$ (acting as the evaluator and verifier) hold a reactive circuit that is defined by a series of Boolean circuits $f_1, \ldots, f_\ell$. For each circuit f_j, $\mathcal{P}$ and $\mathcal{V}$ hold two input vectors $\boldsymbol{x}_j$ and $\boldsymbol{y}_j$ respectively, which may rely on the previous outputs $f_1(\mathsf{state}_0, \boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, f_{j-1}(\mathsf{state}_{j-2}, \boldsymbol{x}_{j-1}, \boldsymbol{y}_{j-1})$. For each $j \in [1, \ell]$, let g_j be the verification circuit corresponding to circuit f_j. Let $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle.

Preprocessing phase. $\mathcal{P}$ and $\mathcal{V}$ do the following:

1. Both parties call functionality $\mathcal{F}_{\mathsf{IZK}}$ to initialize a global key $\Delta \in \{0,1\}^\lambda$ sampled uniformly by $\mathcal{V}$.
2. For $j \in [1, \ell]$, $\mathcal{P}$ runs $(F_j, \mathrm{E}_j^{in}, \mathrm{E}_j^{st}, \mathrm{D}_j) \leftarrow \mathsf{Garble}(f_j, \mathrm{E}_{j-1}^{st})$ where $\mathrm{E}_0^{st} = \bot$, and then sends a garbled circuit F_j to $\mathcal{V}$.

Online evaluation phase. From $j = 1$ to ℓ, $\mathcal{P}$ and $\mathcal{V}$ know the inputs $\boldsymbol{x}_j, \boldsymbol{y}_j \in \{0,1\}^m$, and execute the following steps.

3. Both parties call functionality $\mathcal{F}_{\mathsf{IZK}}$ on input $\boldsymbol{x}_j$ to generate a vector of authenticated bits $[\![\boldsymbol{x}_j]\!]$.
4. $\mathcal{P}$ and $\mathcal{V}$ execute the following steps to let $\mathcal{V}$ obtain the garbled labels on $\boldsymbol{y}_j$.
   (a) $\mathcal{P}$ runs the encoding algorithm two times: $\mathsf{L}[\mathbf{0}_j] \leftarrow \mathsf{Encode}(\mathrm{E}_j, \bot, 0^m)$ and $\mathsf{L}[\mathbf{1}_j] \leftarrow \mathsf{Encode}(\mathrm{E}_j, \bot, 1^m)$, where $\mathsf{L}[\mathbf{0}_j] = (\mathsf{L}[0_{j,1}], \ldots, \mathsf{L}[0_{j,m}])$ and $\mathsf{L}[\mathbf{1}_j] = (\mathsf{L}[1_{j,1}], \ldots, \mathsf{L}[1_{j,m}])$.
   (b) For $i \in [1, m]$, $\mathcal{P}$ (as a sender) and $\mathcal{V}$ (as a receiver) call functionality $\mathcal{F}_{\mathsf{OT}}$ on respective input $(\mathsf{L}[0_{j,i}], \mathsf{L}[1_{j,i}])$ and $y_{j,i} \in \{0,1\}$, and $\mathcal{V}$ obtains $\mathsf{L}[y_{j,i}]$. Then, $\mathcal{V}$ sets $\mathsf{L}[\boldsymbol{y}_j] := (\mathsf{L}[y_{j,1}], \ldots, \mathsf{L}[y_{j,m}])$.
5. $\mathcal{P}$ runs $\mathsf{L}[\boldsymbol{x}_j] \leftarrow \mathsf{Encode}(\mathrm{E}_j, \boldsymbol{x}_j, \bot)$, and sends $\mathsf{L}[\boldsymbol{x}_j]$ to $\mathcal{V}$.
6. $\mathcal{V}$ runs $(\mathsf{L}[\mathsf{state}_j], \mathsf{L}[\boldsymbol{z}_j]) \leftarrow \mathsf{Eval}(F_j, (\mathsf{L}[\mathsf{state}_{j-1}], \mathsf{L}[\boldsymbol{x}_j], \mathsf{L}[\boldsymbol{y}_j]))$ where $\mathsf{L}[\mathsf{state}_0] = \bot$.
7. *Output processing:* $\mathcal{P}$ and $\mathcal{V}$ execute the following to obtain the output of circuit f_j.
   - If $\mathcal{V}$ needs to obtain $\boldsymbol{z}_j$, then $\mathcal{P}$ sends D_j to $\mathcal{V}$ who runs $\boldsymbol{z}_j \leftarrow \mathsf{Decode}(\mathsf{L}[\boldsymbol{z}_j], \mathrm{D}_j)$. In this case, if $\mathcal{P}$ will also get $\boldsymbol{z}_j$ (i.e., $\boldsymbol{z}_j$ is opened), then $\mathcal{V}$ directly sends $\boldsymbol{z}_j$ to $\mathcal{P}$.
   - If only $\mathcal{P}$ will get $\boldsymbol{z}_j$, then $\mathcal{V}$ sends $\mathsf{L}[\boldsymbol{z}_j]$ to $\mathcal{P}$ who runs $\boldsymbol{z}_j \leftarrow \mathsf{Decode}(\mathsf{L}[\boldsymbol{z}_j], \mathrm{D}_j)$.

Online reveal-and-prove phase. $\mathcal{P}$ and $\mathcal{V}$ execute as follows.

8. $\mathcal{V}$ sets $\tau := \mathsf{H}(\mathsf{L}[\boldsymbol{y}_1], \ldots, \mathsf{L}[\boldsymbol{y}_\ell])$ and sends $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_\ell, \tau)$ to $\mathcal{P}$. For $j \in [1, \ell]$, $\mathcal{P}$ runs $\mathsf{L}[\boldsymbol{y}_j] \leftarrow \mathsf{Encode}(e_j, \bot, \boldsymbol{y}_j)$. Then $\mathcal{P}$ computes $\tau' := \mathsf{H}(\mathsf{L}[\boldsymbol{y}_1], \ldots, \mathsf{L}[\boldsymbol{y}_\ell])$ and aborts if $\tau \neq \tau'$.
9. From $j = 1$ to ℓ, $\mathcal{P}$ sets $(\mathsf{state}_j, \hat{\boldsymbol{z}}_j) := f_j(\mathsf{state}_{j-1}, \boldsymbol{x}_j, \boldsymbol{y}_j)$ where $\mathsf{state}_0 = \bot$. For each output $\boldsymbol{z}_j$ obtained by $\mathcal{P}$, it checks $\boldsymbol{z}_j = \hat{\boldsymbol{z}}_j$ and aborts if the check fails.
10. Let $\mathsf{state}_0^* = \bot$. From $j = 1$ to ℓ, $\mathcal{P}$ and $\mathcal{V}$ do the following:
    (a) Both parties call the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on input $[\![\mathsf{state}_{j-1}^*]\!]$, $[\![\boldsymbol{x}_j]\!]$ and circuit g_j to generate $[\![\mathsf{state}_j^*]\!]$ and $[\![\boldsymbol{z}_j^*]\!]$ with $(\mathsf{state}_j^*, \boldsymbol{z}_j^*) = g_j(\mathsf{state}_{j-1}^*, \boldsymbol{x}_j)$.
    (b) If $\mathcal{V}$ outputs $\boldsymbol{z}_j$, then both parties call the (check) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on input $[\![\boldsymbol{z}_j^*]\!] - \boldsymbol{z}_j$ to verify that $\boldsymbol{z}_j = \boldsymbol{z}_j^*$. If $\boldsymbol{z}_j \neq \boldsymbol{z}_j^*$, $\mathcal{V}$ aborts.

Figure 19: **Protocol for securely instantiating $\mathcal{F}_{\mathsf{GP2PC}}$ in the $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{IZK}})$-hybrid model.**

D Instantiation of Functionality $\mathcal{F}_{\mathsf{GP2PC}}$

In this section, we present an efficient protocol to securely instantiate functionality $\mathcal{F}_{\mathsf{GP2PC}}$ (shown in Figure 3), based on a garbling scheme and functionalities $\mathcal{F}_{\mathsf{OT}}$ (shown in Figure 8) and $\mathcal{F}_{\mathsf{IZK}}$ (shown in Figure 11). Before describing the protocol, we give the definition of garbling schemes.

Garbling scheme. Following previous work [BHR12], we define a garbling scheme $\mathcal{GS} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Decode}, \mathsf{Eval})$ that is extended to explicitly consider two-input circuits.

- $(F, \mathrm{E}, \mathrm{D}) \leftarrow \mathsf{Garble}(f)$: On input a circuit f, this algorithm outputs a garbled circuit F, an encoding information $\textsc{e}$ and a decoding information $\textsc{d}$. (Here we omit the input of security parameter 1^λ for simplicity.)

- $(\mathsf{L}[\boldsymbol{x}], \mathsf{L}[\boldsymbol{y}]) \leftarrow \mathsf{Encode}(\textsc{e}, \boldsymbol{x}, \boldsymbol{y})$: On input the encoding information $\textsc{e}$ and two input vectors $\boldsymbol{x}, \boldsymbol{y}$, this algorithm outputs two garbled inputs $\mathsf{L}[\boldsymbol{x}], \mathsf{L}[\boldsymbol{y}]$. If $\boldsymbol{x} = \perp$ (resp., $\boldsymbol{y} = \perp$), then $\mathsf{L}[\boldsymbol{y}] \leftarrow \mathsf{Encode}(\textsc{e}, \perp, \boldsymbol{y})$ (resp., $\mathsf{L}[\boldsymbol{x}] \leftarrow \mathsf{Encode}(\textsc{e}, \boldsymbol{x}, \perp)$).
- $\mathsf{L}[\boldsymbol{z}] \leftarrow \mathsf{Eval}(F, \mathsf{L}[\boldsymbol{x}], \mathsf{L}[\boldsymbol{y}])$: On input a garbled circuit F and garbled inputs $\mathsf{L}[\boldsymbol{x}], \mathsf{L}[\boldsymbol{y}]$, this algorithm outputs a garbled output $\mathsf{L}[\boldsymbol{z}]$.
- $\boldsymbol{z} \leftarrow \mathsf{Decode}(\mathsf{L}[\boldsymbol{z}], \textsc{d})$: On input a garbled output $\mathsf{L}[\boldsymbol{z}]$ and a decoding information $\textsc{d}$, this algorithm outputs $\boldsymbol{z} = f(\boldsymbol{x}, \boldsymbol{y})$.

In the above definition, for a vector $\boldsymbol{x} = (x_1, \ldots, x_m) \in \{0,1\}^m$, $\mathsf{L}[\boldsymbol{x}]$ is defined as $(\mathsf{L}[x_1], \ldots, \mathsf{L}[x_m])$ where $\mathsf{L}[x_i]$ is a garbled label on bit x_i for $i \in [1, m]$. We recall that a garbling scheme $\mathcal{GS}$ satisfies simulation-based privacy [BHR12], if there exists a simulator $\mathcal{S}_{\mathsf{GC}}$, who takes as input $(\boldsymbol{y}, \boldsymbol{z})$ and circuit f, and then outputs $(F', X', Y', \textsc{d}')$ which is computationally indistinguishable from $(F, \mathsf{L}[\boldsymbol{x}], \mathsf{L}[\boldsymbol{y}], \textsc{d})$, where $(F, \textsc{e}, \textsc{d}) \leftarrow \mathsf{Garble}(f)$ and $(\mathsf{L}[\boldsymbol{x}], \mathsf{L}[\boldsymbol{y}]) \leftarrow \mathsf{Encode}(\textsc{e}, \boldsymbol{x}, \boldsymbol{y})$. In addition, we require that for any input vector $\boldsymbol{y}$, every PPT adversary, who is given $(F, \mathsf{L}[\boldsymbol{x}], \mathsf{L}[\boldsymbol{y}], \textsc{d})$ as defined above, cannot produce $(\boldsymbol{y}', Y')$ such that $\boldsymbol{y}' \neq \boldsymbol{y}$ and $Y' = \mathsf{Encode}(\textsc{e}, \perp, \boldsymbol{y}')$. It is not hard to see that the security property holds if the garbling scheme satisfies simulation-based privacy. Simulation-based privacy is satisfied by known garbling schemes, e.g., [Yao86, BMR90, KS08, ZRE15, RR21].

We apply a garbling scheme for a reactive circuit that consists of a series of Boolean circuits $f_1, \ldots, f_\ell$ and maintains a state. Let $\mathsf{state}_0 = \perp$ be the initial state. For $j \in [1, \ell]$, state_{j-1} is used in the computation of circuit f_j, and an updated state state_j is output by f_j, i.e., $(\mathsf{state}_j, \boldsymbol{z}_j) \leftarrow f_j(\mathsf{state}_{j-1}, \boldsymbol{x}_j, \boldsymbol{y}_j)$. To handle this case, we denote by $\textsc{e}_j^{in}$ the encoding information on the inputs of f_j, and use $\textsc{e}_j^{st}$ to represent the encoding information on state_j, i.e., garbled labels $\mathsf{L}[\mathsf{state}_j]$ on state_j can be computed with $\textsc{e}_j^{st}$ and state_j. Informally, we can write $\mathsf{L}[\mathsf{state}_j] \leftarrow \mathsf{Encode}(\textsc{e}_j^{st}, \mathsf{state}_j)$. Therefore, for garbling circuit f_j, $\textsc{e}_{j-1}^{st}$ is also input to Garble, and Garble additionally outputs $\textsc{e}_j^{st}$, i.e., $(F_j, \textsc{e}_j^{in}, \textsc{e}_j^{st}, \textsc{d}_j) \leftarrow \mathsf{Garble}(f_j, \textsc{e}_{j-1}^{st})$. In this way, we can guarantee that the encoding information $\textsc{e}_{j-1}^{st}$ used for garbling f_{j-1} and f_j is identical. When evaluating a reactive circuit, the state information can be transferred by garbled labels on the state. That is, garbled labels $\mathsf{L}[\mathsf{state}_{j-1}]$ are used for the evaluation of garbled circuit F_j, and then garbled labels $\mathsf{L}[\mathsf{state}_j]$ are output and used in the evaluation of garbled circuit F_{j+1}. We denote by $(\mathsf{L}[\mathsf{state}_j], \mathsf{L}[\boldsymbol{z}_j]) \leftarrow \mathsf{Eval}(F_j, (\mathsf{L}[\mathsf{state}_{j-1}], \mathsf{L}[\boldsymbol{x}_j], \mathsf{L}[\boldsymbol{y}_j]))$ such an evaluation procedure. Note that for garbling reactive circuits, we only change the description of algorithms to keep compatible with the definition of reactive circuits, and the security properties as we need are the same.

Protocol for instantiating functionality $\mathcal{F}_{\mathsf{GP2PC}}$. In Figure 19, we present a concretely efficient protocol to instantiate functionality $\mathcal{F}_{\mathsf{GP2PC}}$ (shown in Figure 3) in the $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{IZK}})$-hybrid model. [6] This protocol combines Yao's 2PC protocol [Yao86] based on garbled circuits with the recent interactive ZK proof modeled in functionality $\mathcal{F}_{\mathsf{IZK}}$. For the sake of simplicity, we consider that an output $\boldsymbol{z}_j$ is sent to $\mathcal{P}$, or $\mathcal{V}$, or both of them, and do not split the output $\boldsymbol{z}_j$ such that $\mathcal{P}$ and $\mathcal{V}$ obtain different parts of $\boldsymbol{z}_j$. Our protocol can be straightforwardly extended to support the general case where $\boldsymbol{z}_j$ allows to be split. In protocol Π_{GP2PC}, for $j \in [1, \ell], i \in [1, m]$, $0_{j,i}$ and $1_{j,i}$ are zero and one respectively corresponding to the bit $y_{j,i}$, and $\mathsf{L}[0_{j,i}]$ and $\mathsf{L}[1_{j,i}]$ are 0-label and 1-label respectively associated with garbled label $\mathsf{L}[y_{j,i}]$. The authentication for $\mathcal{V}$'s input $\boldsymbol{y}_j$ is realized by sending $\mathsf{L}[\boldsymbol{y}_j]$ to $\mathcal{P}$. We use a random oracle H to compress the garbled labels $\mathsf{L}[\boldsymbol{y}_1], \ldots, \mathsf{L}[\boldsymbol{y}_\ell]$, which reduces the communication by $(\ell m - 1)\lambda$ bits. When proving circuits, the state information

[6] Here functionality $\mathcal{F}_{\mathsf{IZK}}$ does not include any command in $\mathcal{F}_{\mathsf{GP2PC}}$ (i.e., it now has only the commands associated with IZK based on IT-MACs).

is transferred by IT-MACs on the state, i.e., $[\![\mathsf{state}_j^*]\!]$ with $\mathsf{state}_j^* = \mathsf{state}_j$ for an honest protocol execution. An output $\boldsymbol{z}_j$ obtained by verifier $\mathcal{V}$ is checked using the (check) command of $\mathcal{F}_{\mathsf{IZK}}$.

Optimizations. For most of known efficient garbling schemes with free-XOR optimization [KS08], we have that $\mathsf{lsb}(\mathsf{L}[b]) \oplus \mathsf{lsb}(\mathsf{L}[0]) = b$ for a bit $b \in \{0, 1\}$. When using these garbling schemes, $\mathcal{V}$ can send $\mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j])$ instead of $\mathsf{L}[\boldsymbol{z}_j]$ to $\mathcal{P}$ so that $\mathcal{P}$ obtains the output $\boldsymbol{z}_j = \mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j]) \oplus \mathsf{lsb}(\mathsf{L}[\mathbf{0}_j])$ where $\mathsf{L}[\mathbf{0}_j]$ are a vector of garbled labels known by $\mathcal{P}$. This optimization allows us to reduce the communication by $n(\lambda - 1)$ bits for n the length of vector $\boldsymbol{z}_j$. If the event that $\mathcal{V}$ will output $\boldsymbol{z}_j$ is known in the preprocessing phase, then the decoding information D_j is able to be sent ahead of time in the preprocessing phase. This allows us to reduce one round of communication and the online communication cost.

In some applications (e.g., our main protocol Π_{AuthData} shown in Section 4.2), $\mathcal{V}$ is allowed to obtain a part of $\mathcal{P}$'s inputs in the reveal-then-prove phase (e.g., $\mathcal{P}$'s XOR shares of $h_C = \mathsf{AES}(\mathsf{key}_C, \mathbf{0})$ and $h_S = \mathsf{AES}(\mathsf{key}_S, \mathbf{0})$). For example, if $\boldsymbol{x}_j$ is allowed to be known by $\mathcal{V}$, $\mathcal{P}$ can send $\boldsymbol{x}'_j$ to $\mathcal{V}$, and then both parties call the (check) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on input $[\![\boldsymbol{x}_j]\!] - \boldsymbol{x}'_j$ to verify that $\boldsymbol{x}_j = \boldsymbol{x}'_j$. If $\boldsymbol{x}_j$ was sent before revealing $\mathcal{V}$'s inputs to $\mathcal{P}$, then it can be further optimized. In the reveal-then-prove phase, $\mathcal{P}$ can directly send $\boldsymbol{x}_j$ to $\mathcal{V}$ before receiving $\mathcal{V}$'s inputs, and $\boldsymbol{x}_j$ can be defined in circuit g_j without needing to authenticate $\boldsymbol{x}_j$ with IT-MACs. This would slightly optimize the efficiency of the protocol instantiating $\mathcal{F}_{\mathsf{IZK}}$, and remove the communication round to generate $[\![\boldsymbol{x}_j]\!]$.

Theorem 3. *If the garbling scheme satisfies simulation-based privacy, then protocol Π_{GP2PC} (shown in Figure 19) securely realizes functionality $\mathcal{F}_{\mathsf{GP2PC}}$ (shown in Figure 3) in the $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{IZK}})$-hybrid model, assuming H is a random oracle.*

Proof. We first consider the case of a malicious prover $\mathcal{P}$ and then consider the case of a malicious verifier $\mathcal{V}$. In each case, we construct a probabilistic polynomial time (PPT) simulator $\mathcal{S}$ given access to functionality $\mathcal{F}_{\mathsf{GP2PC}}$, which runs a PPT adversary $\mathcal{A}$ as a subroutine, and emulates functionalities $\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{IZK}}$. Whenever $\mathcal{A}$ or the honest party simulated by $\mathcal{S}$ will abort, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{GP2PC}}$, and then aborts. In both cases, $\mathcal{S}$ simulates a random oracle H by answering a random string in $\{0, 1\}^\lambda$ for each query while keeping the consistency of answers. For the case of a malicious $\mathcal{V}$, $\mathcal{S}$ also invokes a simulator $\mathcal{S}_{\mathsf{GC}}$ for the underlying garbling scheme. Since our protocol sends a garbled circuit in the preprocessing phase and then evaluates it in the online phase, we require that $\mathcal{S}_{\mathsf{GC}}$ is adaptive, i.e., $\mathcal{S}_{\mathsf{GC}}(f)$ first outputs a garbled circuit F' and a decoding information D' as well as an additional state information ST, and then $\mathcal{S}_{\mathsf{GC}}(\mathrm{ST}, \boldsymbol{y}, \boldsymbol{z})$ outputs an encoded input (X', Y'). Note that known garbling schemes satisfy adaptive security in the random-oracle model. When $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{GC}}$, the description of $\mathcal{S}_{\mathsf{GC}}$ is naturally extended to handle the case of reactive circuits (see below for details).

Malicious prover. By emulating ideal functionalities $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{IZK}}$, $\mathcal{S}$ interacts with $\mathcal{A}$ as follows.

1. For $j \in [1, \ell]$, $\mathcal{S}$ receives a garbled circuit F_j from $\mathcal{A}$.
2. From $j = 1$ to ℓ, $\mathcal{S}$ simulates the online evaluation phase as follows:
   (a) $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving $\boldsymbol{x}_j$ and the MAC tags on $[\![\boldsymbol{x}_j]\!]$ from $\mathcal{A}$. Then, $\mathcal{S}$ sends $\boldsymbol{x}_j$ to functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as $\mathcal{P}$'s input.
   (b) $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{OT}}$ by receiving garbled labels $\mathsf{L}[\mathbf{0}_j]$ and $\mathsf{L}[\mathbf{1}_j]$ from $\mathcal{A}$.
   (c) $\mathcal{S}$ receives garbled labels $\mathsf{L}[\boldsymbol{x}_j]$ from $\mathcal{A}$.
   (d) If $\mathcal{V}$ would obtain an output $\boldsymbol{z}_j$, $\mathcal{S}$ receives a decoding information D_j from $\mathcal{A}$.
   (e) $\mathcal{S}$ defines a circuit f'_j as follows:

    - On input a state $\mathsf{L}[\mathsf{state}_{j-1}]$ and $\mathcal{V}$'s input $\boldsymbol{y}_j$, set $\mathsf{L}[\boldsymbol{y}_j]$ according to $(\mathsf{L}[\mathbf{0}_j], \mathsf{L}[\mathbf{1}_j])$.
    - Run $(\mathsf{L}[\mathsf{state}_j], \mathsf{L}[\boldsymbol{z}_j]) \leftarrow \mathsf{Eval}(F_j, (\mathsf{L}[\mathsf{state}_{j-1}], \mathsf{L}[\boldsymbol{x}_j], \mathsf{L}[\boldsymbol{y}_j]))$.
    - If $\mathcal{V}$ would obtain an output $\boldsymbol{z}_j$, then run $\boldsymbol{z}_j \leftarrow \mathsf{Decode}(\mathsf{L}[\boldsymbol{z}_j], \textsc{d}_j)$ and output $(\mathsf{L}[\mathsf{state}_j], \boldsymbol{z}_j)$. Otherwise, output $(\mathsf{L}[\mathsf{state}_j], \mathsf{L}[\boldsymbol{z}_j])$.

   (f) $\mathcal{S}$ sends f'_j to functionality $\mathcal{F}_{\mathsf{GP2PC}}$.

   (g) If both $\mathcal{P}$ and $\mathcal{V}$ would obtain an output $\boldsymbol{z}_j$, $\mathcal{S}$ receives $\boldsymbol{z}_j$ from $\mathcal{F}_{\mathsf{GP2PC}}$, and then sends it to $\mathcal{A}$. If only $\mathcal{P}$ would get the output, $\mathcal{S}$ receives $\mathsf{L}[\boldsymbol{z}_j]$ from $\mathcal{F}_{\mathsf{GP2PC}}$ and sends it to $\mathcal{A}$.

3. In the reveal-and-prove phase, $\mathcal{S}$ receives $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_\ell$ from functionality $\mathcal{F}_{\mathsf{GP2PC}}$. For $j \in [1, \ell]$, $\mathcal{S}$ sets $\mathsf{L}[\boldsymbol{y}_j]$ according to $(\mathsf{L}[\mathbf{0}_j], \mathsf{L}[\mathbf{1}_j])$. Then, $\mathcal{S}$ sends $\tau = \mathsf{H}(\mathsf{L}[\boldsymbol{y}_1], \ldots, \mathsf{L}[\boldsymbol{y}_\ell])$ to $\mathcal{A}$.

4. From $j = 1$ to ℓ, $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving the MAC tags on $\llbracket \mathsf{state}^*_j \rrbracket$ and $\llbracket \boldsymbol{z}^*_j \rrbracket$ from $\mathcal{A}$. Then $\mathcal{S}$ receives $\mathsf{res}_j \in \{\mathsf{true}, \mathsf{false}\}$ from functionality $\mathcal{F}_{\mathsf{GP2PC}}$ and aborts if $\mathsf{res}_j = \mathsf{false}$.

The simulation of $\mathcal{F}_{\mathsf{IZK}}$ is perfect, as $\mathcal{S}$ only receives $\boldsymbol{x}_j$ for $j \in [1, \ell]$ and MAC tags. $\mathcal{S}$ also perfectly simulates functionality $\mathcal{F}_{\mathsf{OT}}$, as it only records garbled labels $(\mathsf{L}[\mathbf{0}_j], \mathsf{L}[\mathbf{1}_j])$ for $j \in [1, \ell]$. The definition of circuit f'_j for each $j \in [1, \ell]$ behaves just as that $\mathcal{V}$ evaluates garbled circuit F_j with the Eval algorithm. The circuits $f'_1, \ldots, f'_\ell$ maintain the state that consists of the corresponding garbled labels. For each $j \in [1, \ell]$, if $\mathcal{V}$ would obtain the output, then the decoding information $\textsc{d}_j$ is received from $\mathcal{A}$, and thus f'_j could output $\boldsymbol{z}_j$ by running the Decode algorithm; otherwise, circuit f'_j just outputs a vector of garbled labels $\mathsf{L}[\boldsymbol{z}_j]$. In this way, honest verifier $\mathcal{V}$ would obtain the same output in both ideal-world execution and real-world execution. Functionality $\mathcal{F}_{\mathsf{GP2PC}}$ sends $\mathsf{L}[\boldsymbol{z}_j]$ to $\mathcal{S}$ as $\mathcal{P}$'s output, and $\mathcal{S}$ forwards it to $\mathcal{A}$. Here $\mathsf{L}[\boldsymbol{z}_j]$ sent by $\mathcal{F}_{\mathsf{GP2PC}}$ has the same distribution as that in the real protocol execution. It is easy to see that τ simulated by $\mathcal{S}$ has the identical distribution as that sent by $\mathcal{V}$ in the real protocol execution. For $j \in [1, \ell]$, state^*_j and $\boldsymbol{z}^*_j$ are computed in the same way in both worlds. Therefore, the check of $\boldsymbol{z}_j = \boldsymbol{z}^*_j$ is identical in both worlds. Overall, the joint distribution of the outputs of honest $\mathcal{V}$ and $\mathcal{A}$ in the real-world execution is identical to that of $\mathcal{V}$ and $\mathcal{S}$ in the ideal-world execution.

Malicious verifier. By emulating functionalities $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{IZK}}$, and invoking the simulator $\mathcal{S}_{\mathsf{GC}}$, $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows.

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{IZK}}$ by receiving $\Delta \in \{0,1\}^\lambda$ from $\mathcal{A}$.

2. For $j \in [1, \ell]$, $\mathcal{S}$ runs $(\textsc{st}_j, F_j, \textsc{d}_j) \leftarrow \mathcal{S}_{\mathsf{GC}}(\textsc{st}_{j-1}, f_j)$ where $\textsc{st}_0 = \perp$. Then it sends a garbled circuit F_j to $\mathcal{A}$.

3. From $j = 1$ to ℓ, $\mathcal{S}$ simulates the online evaluation phase.

   (a) $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving the keys on $\llbracket \boldsymbol{x}_j \rrbracket$ from $\mathcal{A}$.

   (b) $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{OT}}$ by receiving a vector of choice bits $\boldsymbol{y}_j$ from $\mathcal{A}$. Then, $\mathcal{S}$ sends $\boldsymbol{y}_j$ to functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as $\mathcal{V}$'s input.

   (c) If $\mathcal{V}$ will get an output $\boldsymbol{z}_j$, then $\mathcal{S}$ receives $\boldsymbol{z}_j$ from functionality $\mathcal{F}_{\mathsf{GP2PC}}$. Otherwise, $\mathcal{S}$ computes $(\mathsf{state}'_j, \boldsymbol{z}_j) := f_j(\mathsf{state}'_{j-1}, \mathbf{0}, \boldsymbol{y}_j)$ where $\mathsf{state}'_0 = \perp$. Then, $\mathcal{S}$ runs $(\textsc{st}_{\ell+j}, X_j, Y_j) \leftarrow \mathcal{S}_{\mathsf{GC}}(\textsc{st}_{\ell+j-1}, \boldsymbol{y}_j, \boldsymbol{z}_j)$.

   (d) $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{OT}}$ and sends Y_j to $\mathcal{A}$ as the garbled labels on $\boldsymbol{y}_j$.

   (e) $\mathcal{S}$ sends X_j to $\mathcal{A}$ as the garbled labels on unknown input $\boldsymbol{x}_j$.

   (f) For output processing, $\mathcal{S}$ simulates as follows:
    - If $\mathcal{V}$ will obtain an output $\boldsymbol{z}_j$, $\mathcal{S}$ sends $\textsc{d}_j$ to $\mathcal{A}$.
    - If both parties will obtain the output, $\mathcal{S}$ receives $\boldsymbol{z}'_j$ from $\mathcal{A}$. Then, $\mathcal{S}$ computes a vector of errors $\boldsymbol{e}_j := \boldsymbol{z}'_j \oplus \boldsymbol{z}_j$ and sends $\boldsymbol{e}_j$ to functionality $\mathcal{F}_{\mathsf{GP2PC}}$.

- If only $\mathcal{P}$ will obtain the output, $\mathcal{S}$ receives Z_j from $\mathcal{A}$, and runs $\boldsymbol{z}'_j \leftarrow \mathsf{Decode}(Z_j, \mathtt{D}_j)$. Then $\mathcal{S}$ computes $\boldsymbol{e}_j := \boldsymbol{z}'_j \oplus \boldsymbol{z}_j$ and sends $\boldsymbol{e}_j$ to functionality $\mathcal{F}_{\mathsf{GP2PC}}$.

4. In the online reveal-and-prove phase, $\mathcal{S}$ receives a tuple $(\boldsymbol{y}'_1, \ldots, \boldsymbol{y}'_\ell, \tau)$ from $\mathcal{A}$. Then, $\mathcal{S}$ computes $\tau' := \mathsf{H}(Y_1, \ldots, Y_\ell)$, and checks that $\tau = \tau'$ and $\boldsymbol{y}_i = \boldsymbol{y}'_i$ for all $i \in [1, \ell]$. If the check fails, $\mathcal{S}$ aborts.
5. Honest prover $\mathcal{P}$ checks the correctness of all its outputs, and sends abort to functionality $\mathcal{F}_{\mathsf{GP2PC}}$ if the check fails. If $\mathcal{S}$ receives abort from $\mathcal{F}_{\mathsf{GP2PC}}$, then $\mathcal{S}$ aborts.
6. Following the protocol specification, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{IZK}}$ by receiving the keys of IT-MACs from $\mathcal{A}$ and sending true to $\mathcal{A}$ for each call of the (check) command.

It is clear that the simulation of functionality $\mathcal{F}_{\mathsf{IZK}}$ is perfect. From the simulation-based privacy of the garbling scheme, we have that garbled circuits F_j for $j \in [1, \ell]$ simulated by $\mathcal{S}_{\mathsf{GC}}$ are computationally indistinguishable from that sent in the real protocol execution. If $\mathcal{V}$ will obtain an output $\boldsymbol{z}_j$, then $\boldsymbol{z}_j$ is received from functionality $\mathcal{F}_{\mathsf{GP2PC}}$ and the same as the actual output. Otherwise, $\boldsymbol{z}_j$ is computed by $\mathcal{S}$ with $\mathcal{P}$'s input $\mathbf{0}$ and $\mathcal{V}$'s input $\boldsymbol{y}_j$. This does not matter as $\mathcal{A}$ cannot obtain the output in this case. Based on the simulation-based privacy, for each $j \in [1, \ell]$, (X_j, Y_j) simulated by $\mathcal{S}_{\mathsf{GC}}$ is computationally indistinguishable from $(\mathsf{L}[\boldsymbol{x}_j], \mathsf{L}[\boldsymbol{y}_j])$ sent in the real protocol execution. For $j \in [1, \ell]$, if $\mathcal{P}$ needs to get the output $\boldsymbol{z}_j$, then $\mathcal{S}$ extracts an error vector $\boldsymbol{e}_j$ from the message sent by $\mathcal{A}$ and sends it to functionality $\mathcal{F}_{\mathsf{GP2PC}}$. Therefore, the outputs obtained by honest prover $\mathcal{P}$ have the identical distribution in both worlds.

If there exists some $j^* \in [1, \ell]$ such that $\boldsymbol{y}'_{j^*} \neq \boldsymbol{y}_{j^*}$, the real-world execution does not abort if $\tau = \mathsf{H}(\mathsf{L}[\boldsymbol{y}'_1], \ldots, \mathsf{L}[\boldsymbol{y}'_\ell])$ where $\mathsf{L}[\boldsymbol{y}'_j] \leftarrow \mathsf{Encode}(e_j, \perp, \boldsymbol{y}'_j)$ for $j \in [1, \ell]$, while the ideal-world execution always aborts. For $j \in [1, \ell]$, $\boldsymbol{y}'_j$ represents the vector sent from $\mathcal{A}$ in step 8, while $\boldsymbol{y}_j$ denotes the actual input of $\mathcal{V}$. Below, we show that the probability that there exists some $j^* \in [1, \ell]$ such that $\boldsymbol{y}'_{j^*} \neq \boldsymbol{y}_{j^*}$ and the real protocol execution does not abort is negligible in λ. If $\mathcal{A}$ does not make a query $(\mathsf{L}[\boldsymbol{y}'_1], \ldots, \mathsf{L}[\boldsymbol{y}'_\ell])$ to random oracle H, then $\tau = \mathsf{H}(\mathsf{L}[\boldsymbol{y}'_1], \ldots, \mathsf{L}[\boldsymbol{y}'_\ell])$ is uniform in λ. In this case, the probability that $\mathcal{A}$ learns τ is at most $1/2^\lambda$. Otherwise (i.e., $\mathcal{A}$ made a query $(\mathsf{L}[\boldsymbol{y}'_1], \ldots, \mathsf{L}[\boldsymbol{y}'_\ell])$), we can use $\mathcal{A}$ to break the security property of the garbling scheme. In particular, the reduction could retrieve the queries whose outputs of H are equal to τ, and finds which one is the query $(\mathsf{L}[\boldsymbol{y}'_1], \ldots, \mathsf{L}[\boldsymbol{y}'_\ell])$. Then, the reduction sends $(\boldsymbol{y}'_{j^*}, \mathsf{L}[\boldsymbol{y}'_{j^*}])$ to its security experiment. Under the assumption that the garbling scheme satisfies simulation-based privacy, the probability that $\mathcal{A}$ made a query $(\mathsf{L}[\boldsymbol{y}'_1], \ldots, \mathsf{L}[\boldsymbol{y}'_\ell])$ is negligible in λ. Therefore, if the real protocol execution does not abort, then $\boldsymbol{y}'_j = \boldsymbol{y}_j$ for all $j \in [1, \ell]$. Based on the simulation-based privacy, we further have that using Y_j for $j \in [1, \ell]$ to check τ is computationally indistinguishable from using $\mathsf{L}[\boldsymbol{y}_j]$ to check τ, under the condition that $\boldsymbol{y}'_j = \boldsymbol{y}_j$ for all $j \in [1, \ell]$.

In both worlds, honest prover $\mathcal{P}$ always checks the correctness of all its outputs. If an error chosen by $\mathcal{A}$ is introduced to $\mathcal{P}$'s output, the protocol execution would abort in both worlds. In other words, if the protocol execution does not abort, the outputs obtained by $\mathcal{P}$ are correct in both worlds. If the real protocol execution does not abort, then $\boldsymbol{y}'_j = \boldsymbol{y}_j$ for all $j \in [1, \ell]$ as analyzed above. Therefore, in the real-world execution, $\mathcal{V}$ would always receive true when calling the (check) command of functionality $\mathcal{F}_{\mathsf{IZK}}$. Overall, the joint distribution of the outputs of honest prover $\mathcal{P}$ and $\mathcal{A}$ in the real-world execution is computationally indistinguishable from that of $\mathcal{P}$ and $\mathcal{S}$ in the ideal-world execution. □

E Extending Our Protocol for TLS 1.3

In this section, we extend our protocol for TLS 1.3. The main differences between TLS 1.2 and TLS 1.3 are the handshake phase and the key derivation function.

The TLS 1.3 handshake process involves only one roundtrip. The client sends the randomness together with the ephemeral public key for key exchange to the server. Once receives this request message, the server chooses his randomness and ephemeral public key, and then generates the shared pre-master key. The server encrypts remaining messages in the handshake phase via a secret key derived from the pre-master key. Fortunately, this secret key is independent of other derived keys. We can just open it to $\mathcal{P}$ in our 2PC protocol, $\mathcal{P}$ could decrypt the handshake message in plain.

The key derivation function in TLS 1.3 is based on the HMAC-based Key Derivation Function (HKDF) [Kra10, KE10]. The HKDF function adopts two sub-functions: $\mathsf{HKDF.Extract}$ and $\mathsf{HKDF.Expand}$. The $\mathsf{HKDF.Extract}$ function extracts a pseudorandom key (okey) from the input key material (ikey) with a non-secret random *salt* by applying the HMAC function, i.e., $\mathsf{okey} = \mathsf{HKDF.Extract}(salt, \mathsf{ikey}) = \mathsf{HMAC}(salt, \mathsf{ikey})$. Note that *salt* is the HMAC key and ikey is the HMAC input.

The okey is then expanded to secret keys with desired length. The $\mathsf{HKDF.Expand}$ function with output length ℓ is defined below:

$$\mathsf{HKDF.Expand}_\ell = T_1 \,\|\, \cdots \,\|\, T_{n-1} \,\|\, \mathsf{Trunc}_m(T_n),$$

where T_0 is the empty string, $T_i = \mathsf{HMAC}(\mathsf{okey}, T_{i-1} \| msg \| i)$ for each $i \in [1, n]$, $n = \lceil \ell/256 \rceil$ and $m = \ell - 256 \cdot (n-1)$. It is easy to see that all optimizations for PRF in TLS 1.2 can be directly extended to the key derivation function in TLS 1.3.

F Proof of Security for Our Main Protocol

In this section, we give the detailed proof of Theorem 1. This theorem is restated as follows.

Theorem 4 (Theorem 1, restated)**.** *If the decisional Diffie-Hellman (DDH) assumption holds, then protocol Π_{AuthData} (shown in Figures 4 and 5) securely realizes functionality $\mathcal{F}_{\mathsf{AuthData}}$ (shown in Figure 1) 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.*

Proof. We first consider the case of a malicious $\mathcal{V}$ and then consider the case of a malicious $\mathcal{P}$. In each case, we construct a PPT simulator $\mathcal{S}$ given access to functionality $\mathcal{F}_{\mathsf{AuthData}}$, which runs a PPT adversary $\mathcal{A}$ as a subroutine, and emulates functionalities $\mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{IZK}}, \mathcal{F}_{\mathsf{Conv}}$. In the proof, we use $\mathcal{S}$ to denote a simulator instead of a TLS server, and $\mathcal{S}$ plays the role of the server to interact with $\mathcal{A}$. During the protocol execution, we always consider that $\mathcal{A}$ can intercept and tamper all TLS messages transmitted between $\mathcal{P}$ and the server, even if $\mathcal{A}$ only corrupts $\mathcal{V}$ and $\mathcal{P}$ is honest. Whenever $\mathcal{A}$ or the honest party simulated by $\mathcal{S}$ will abort, $\mathcal{S}$ sends abort to functionality $\mathcal{F}_{\mathsf{AuthData}}$, and then aborts. We construct $\mathcal{S}$ to simulate four phases of protocol Π_{AuthData}: preprocessing phase, handshake phase, record phase and post-record phase.

Before describing simulator $\mathcal{S}$, we first show three sub-simulators $\mathcal{S}_{\mathsf{E2F}}$, $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$ for sub-protocols Π_{E2F}, Π_{PRF} and Π_{AEAD} respectively. Then $\mathcal{S}$ would invoke these sub-simulators as its subroutine. In this way, our proof would be more modular and clear. For the construction of each sub-simulator, we will use the notation underlying these sub-protocols independently for ease of comprehension.

Simulation of random oracle and ideal cipher. $\mathcal{S}$ simulates a random oracle f_{H} by responding the queries made by $\mathcal{A}$ with uniform strings in $\{0,1\}^{256}$ while keeping the consistency of responses, where f_{H} is used as the compression function of H. Similarly, $\mathcal{S}$ simulates an ideal cipher AES by responding every key-message query made by $\mathcal{A}$ with random strings in $\{0,1\}^{128}$ while keeping the consistency of responses.

Malicious prover $\mathcal{P}$. We first give the constructions of $\mathcal{S}_{\mathsf{E2F}}$, $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$ for three sub-protocols, and then describe the construction of $\mathcal{S}$ for main protocol Π_{AuthData}. These simulators emulate the same functionalities $\mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{IZK}}, \mathcal{F}_{\mathsf{Conv}}$, and only $\mathcal{S}$ is given access to functionality $\mathcal{F}_{\mathsf{AuthData}}$.

Simulation and analysis of sub-protocol Π_{E2F}. By emulating functionality $\mathcal{F}_{\mathsf{OLEe}}$, $\mathcal{S}_{\mathsf{E2F}}$ interacts with a PPT adversary $\mathcal{A}$ as follows.

1. $\mathcal{S}_{\mathsf{E2F}}$ emulates functionality $\mathcal{F}_{\mathsf{OLEe}}$ by receiving input $(a_1, b_1, a_1, b_1', r_1)$, error vectors $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_5$ and $\mathcal{P}$'s shares of $[a_1b_2]_p, [a_2b_1]_p, [a_1b_2']_p, [a_2b_1']_p, [r_1r_2]_p$. Then, $\mathcal{S}_{\mathsf{E2F}}$ computes $e_1 = (\boldsymbol{g} * \boldsymbol{e}_1) \odot \boldsymbol{b}_2$, $e_2 = (\boldsymbol{g} * \boldsymbol{e}_2) \odot \boldsymbol{a}_2$, $e_3 = (\boldsymbol{g} * \boldsymbol{e}_3) \odot \boldsymbol{b}_2'$, $e_4 = (\boldsymbol{g} * \boldsymbol{e}_4) \odot \boldsymbol{a}_2$ and $e_5 = (\boldsymbol{g} * \boldsymbol{e}_5) \odot \boldsymbol{r}_2$, where $\boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{b}_2', \boldsymbol{r}_2$ are the bit-decomposition of field elements $a_2, b_2, b_2', r_2 \in \mathbb{Z}_p$.
2. $\mathcal{S}_{\mathsf{E2F}}$ computes the shares of $[c]_p$, $[c']_p$ and $[r^2]_p$ held by $\mathcal{P}$ following the protocol specification. In addition, $\mathcal{S}_{\mathsf{E2F}}$ computes $e_c := e_1 + e_2$, $e_{c'} := e_3 + e_4$ and $e_r := 2e_5$.
3. On behalf of honest $\mathcal{V}$, $\mathcal{S}_{\mathsf{E2F}}$ simulates the Open procedure on the values $\epsilon_1, \epsilon_2, \epsilon_3, w$ opened, following the protocol description. Here, $\mathcal{S}_{\mathsf{E2F}}$ could know the point (x_2, y_2) according to the construction of the simulator in the proof of Theorem 1, and can sample $a_2, b_2, b_2', r_2 \leftarrow \mathbb{Z}_p$. Then $\mathcal{S}_{\mathsf{E2F}}$ is able to compute $\mathcal{V}$'s shares of all additive sharings used in protocol Π_{E2F}.
4. During the Open procedure on the values $\epsilon_1, \epsilon_2, \epsilon_3, w$, $\mathcal{S}_{\mathsf{E2F}}$ extracts the errors e_6, e_7, e_8, e_9 by computing that the received value minuses the value should be sent.
5. $\mathcal{S}_{\mathsf{E2F}}$ is able to compute an error $e_z = f(a, r, e_c, e_{c'}, e_r, e_6, \ldots, e_9)$, which is added into the secret on $[z]_p$, where f is a function depending on the definition of z.

To analyze the security for the case of a malicious prover $\mathcal{P}$, we put sub-protocol Π_{E2F} into the main protocol Π_{AuthData}. In this case, an extra error e_0 introduced by $\mathcal{A}$ in the main protocol Π_{AuthData} would be added into the secret on $[z]_p = [\widetilde{\mathsf{pms}}]_p$. Thus, the error is now changed as $e' = f(a, r, e_c, e_{c'}, e_r, e_6, \ldots, e_9) + e_0$. According to the analysis for Π_{AuthData} shown in Appendix F, e' has to be zero and otherwise would incur the protocol execution aborts. This makes e_c, e_6, e_7, e_8 that are multiplied with a be zero, since a is uniform in $\mathbb{Z}_p$ and has a sufficiently high entropy. According to the DDH assumption, $Z_2 = (x_2, y_2) = t_{\mathcal{V}} \cdot T_S$ is computationally indistinguishable from a uniform group element. This means that $r = \eta - \epsilon_3 \in \mathbb{Z}_p$ with $\eta = (y_2 - y_1)/(x_2 - x_1) \in \mathbb{Z}_p$ has a sufficiently high entropy. Thus, $e_{c'}, e_r, e_9$ that are multiplied with r must be zero if the protocol does not abort according to the definition of f. We have $e_c, e_{c'}, e_r = 0$, which means $(\boldsymbol{g} * \boldsymbol{e}_1) \odot \boldsymbol{b}_2 + (\boldsymbol{g} * \boldsymbol{e}_2) \odot \boldsymbol{a}_2 = 0$, $(\boldsymbol{g} * \boldsymbol{e}_3) \odot \boldsymbol{b}_2' + (\boldsymbol{g} * \boldsymbol{e}_4) \odot \boldsymbol{a}_2 = 0$ and $(\boldsymbol{g} * \boldsymbol{e}_5) \odot \boldsymbol{r}_2 = 0$. This allows $\mathcal{A}$ to guess a few bits of $\boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{b}_2', \boldsymbol{r}_2$, where an incorrect guess would incur the main protocol execution aborts. Therefore, $\mathcal{A}$ could reveal at most one-bit information on average for (a_2, b_2, b_2', r_2). From $x_2 = \epsilon_1 + b_2 + b_1 + x_1$, $y_2 = \epsilon_2 + b_2' + b_1' + y_1$ and $\eta = (y_2 - y_1)/(x_2 - x_1) = \epsilon_3 + r_2 + r_1$, we have that $\mathcal{A}$ could reveal at most one-bit information on (x_2, y_2) or z. This is harmless as $Z_2 = (x_2, y_2)$ has a sufficiently high entropy.

Simulation and analysis of sub-protocol Π_{PRF}. By emulating functionality $\mathcal{F}_{\mathsf{GP2PC}}$, $\mathcal{S}_{\mathsf{PRF}}$ interacts with a PPT adversary $\mathcal{A}$.

1. $\mathcal{S}_{\mathsf{PRF}}$ is given secret' from the simulation of the main protocol Π_{AuthData} in the handshake phase. $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving a circuit $\widetilde{\mathcal{C}}_1$ from $\mathcal{A}$ and computing $(IV_1', IV_2') :=$

$\widetilde{\mathcal{C}_1}(\mathsf{secret}')$. Then, $\mathcal{S}_{\mathsf{PRF}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by sending IV'_1 to $\mathcal{A}$.

2. From $i = 1$ to n, $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving a circuit $\widetilde{\mathcal{C}_{2,i}}$ from $\mathcal{A}$ and computing $M'_i := \widetilde{\mathcal{C}_{2,i}}(IV'_2)$, where input W'_i is defined in $\widetilde{\mathcal{C}_{2,i}}$. Then, $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by sending M'_i to $\mathcal{A}$.
3. $\mathcal{S}_{\mathsf{PRF}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving a circuit $\widetilde{\mathcal{C}_3}$ from $\mathcal{A}$ and computing $\mathsf{der}' := \widetilde{\mathcal{C}_3}(IV'_2) \in \{0,1\}^{\ell}$, where inputs $X'_1, \ldots, X'_n$ chosen by $\mathcal{A}$ have been defined in $\widetilde{\mathcal{C}_3}$.
4. If type = "open", $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by sending der' to $\mathcal{A}$. If type = "partial open", $\mathcal{S}_{\mathsf{PRF}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by parsing $\mathsf{der}' = (\mathsf{key}'_C, IV'_C, \mathsf{key}'_S, IV'_S)$ and sending (IV'_C, IV'_S) to $\mathcal{A}$.
5. In the post-record phase, $\mathcal{S}_{\mathsf{PRF}}$ is given secret^* from the simulation of the main protocol Π_{AuthData}. $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by generating $IV^*_2 = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \text{opad})$, and sending false to $\mathcal{A}$ and aborting if $IV'_1 \neq f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \text{ipad})$.
6. $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $M^*_i = f_{\mathsf{H}}(IV^*_2, W^*_i)$ for each $i \in [1, n]$ where $W^*_i = f_{\mathsf{H}}(IV^*_1, M^*_{i-1})$ and $M^*_0 = \mathit{label}\|\mathit{msg}$, and then sending false to $\mathcal{A}$ and aborting if $M'_i \neq M^*_i$ for any $i \in [1, n]$.
7. $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $\mathsf{der}^* := \big(f_{\mathsf{H}}(IV^*_2, X^*_1), \ldots, f_{\mathsf{H}}(IV^*_2, X^*_{n-1}), \mathsf{Trunc}_m(f_{\mathsf{H}}(IV^*_2, X^*_n))\big)$ for every public value $X^*_i := f_{\mathsf{H}}(IV^*_1, M^*_i\|\mathit{label}\|\mathit{msg})$. If type = "open", then $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by sending false to $\mathcal{A}$ if $\mathsf{der}^* \neq \mathsf{der}'$ or true otherwise. If type = "partial open", then $\mathcal{S}_{\mathsf{PRF}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by sending false to $\mathcal{A}$ if $IV^*_C \neq IV'_C$ or $IV^*_S \neq IV'_S$ or true otherwise.

To analyze the security for the case of a malicious prover $\mathcal{P}$, we cast sub-protocol Π_{PRF} into the main protocol Π_{AuthData}. It is clear that the simulation of random oracle f_{H} and the protocol execution in the handshake phase is perfect. In the post-record phase, it is easy to see that the simulation to check IV'_1 is perfect. By induction, we have that the check of M'_i for all $i \in [1, n]$ simulated by $\mathcal{S}_{\mathsf{PRF}}$ is the same that in the real protocol execution, where $W_i = f_{\mathsf{H}}(IV'_1, M'_{i-1}) = W^*_i$ from the induction and $M'_0 = M^*_0 = \mathit{label}\|\mathit{msg}$. If the protocol execution does not abort, then IV'_1 and M'_i for each $i \in [1, n]$ are computed correctly, and thus $X_i = f_{\mathsf{H}}(IV'_1, M'_i\|\mathit{label}\|\mathit{msg}) = X^*_i$ for each $i \in [1, n]$. Therefore, the simulation of checking der' if type = "open" and checking IV'_C, IV'_S if type = "partial open" is perfect. Overall, $\mathcal{S}_{\mathsf{PRF}}$ perfectly simulates the post-record phase of sub-protocol Π_{PRF}.

If the protocol execution does not abort, all the values opened are correctly computed with secret^* that is a correct secret. In this case, we show that $\mathcal{A}$ cannot learn secret^* for all three cases of type, der^* if type = "secret" and $(\mathsf{key}^*_C, \mathsf{key}^*_S)$ if type = "partial open" in the handshake phase. In the random-oracle model, $\mathcal{A}$ cannot learn secret^* from $IV'_1 = IV^*_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \text{ipad})$ as secret^* has a sufficiently high entropy. Similarly, $\mathcal{A}$ cannot learn $IV^*_2 = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \text{opad})$ from IV'_1 where ipad $\neq$ opad. In other worlds, $IV^*_2 \in \{0,1\}^{256}$ is uniform against the adversary's view. In the handshake phase, $\mathcal{A}$ obtains $M'_i = M^*_i = f_{\mathsf{H}}(IV^*_2, W^*_i)$ for $i \in [1, n]$. In addition, $\mathcal{A}$ gets $\mathsf{der}' = \mathsf{der}^* = \big(f_{\mathsf{H}}(IV^*_2, X^*_1), \ldots, f_{\mathsf{H}}(IV^*_2, X^*_{n-1}), \mathsf{Trunc}_m(f_{\mathsf{H}}(IV^*_2, X^*_n))\big)$ if type = "open", or $(IV'_C, IV'_S) = (IV^*_C, IV^*_S)$ that is a part of der^* if type = "partial open". All the values do not reveal IV^*_2 to $\mathcal{A}$ in the random-oracle model. Therefore, if type = "secret", then $\mathsf{der}^* = \mathsf{PRF}_{\ell}(\mathsf{secret}^*, \mathit{label}, \mathit{msg})$ is kept secret against $\mathcal{A}$, as secret^* and IV^*_2 are unknown for $\mathcal{A}$, and $(\mathit{label}, \mathit{msg})$ is different from that for other two cases of type. If type = "partial open", $\mathcal{A}$ cannot learn $(\mathsf{key}^*_C, \mathsf{key}^*_S)$ as f_{H} is a random oracle, and both of secret^* and IV^*_2 have sufficiently high entropy. Overall, in the handshake phase, these secret values are kept unknown against $\mathcal{A}$.

In the following, we show that $\mathsf{secret}' = \mathsf{secret}^*$ and $IV'_2 = IV^*_2$ if the protocol execution does not abort. If $\mathsf{secret}' \neq \mathsf{secret}^*$, then $IV^*_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \text{ipad}) \in \{0,1\}^{256}$ is uniformly random in the handshake phase as secret^* has a sufficiently high entropy, and the probability that $IV_1' = IV_1^*$ is negligible (i.e., the probability that $\mathcal{A}$ succeeds to guess IV_1^* is negligible). Similarly, if $IV_2' \neq IV_2^*$, then for any $i \in [1, n]$, $M_i^* = f_{\mathsf{H}}(IV_2^*, W_i^*)$ is uniform in the handshake phase, and the probability that $M_i' = M_i^*$ is negligible. In conclusion, if the protocol execution does not abort, then $\mathsf{secret}' = \mathsf{secret}^*$ and $IV_2' = IV_2^*$.

Simulation and analysis of sub-protocol Π_{AEAD}. We construct the simulator $\mathcal{S}_{\mathsf{AEAD}}$ for all four cases, even though the case of $\mathsf{type}_1 =$ "decryption" and $\mathsf{type}_2 =$ "secret" is *not* necessary for the simulation of main protocol Π_{AuthData}. This case is useful for security of the extension of multiple query-response sessions shown in Section 4.3. By emulating functionalities $\mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{Com}}$, simulator $\mathcal{S}_{\mathsf{AEAD}}$ interacts with a PPT adversary $\mathcal{A}$.

1. In the preprocessing phase, $\mathcal{S}_{\mathsf{AEAD}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving $h_{\mathcal{P}}, \mathrm{z}_{0,\mathcal{P}} \in \{0,1\}^{128}$ from $\mathcal{A}$ if $\mathsf{type}_2 =$ "open" or $\mathrm{z}_{0,\mathcal{P}}, \mathrm{z}_{1,\mathcal{P}}, \ldots, \mathrm{z}_{n,\mathcal{P}} \in \{0,1\}^{128}$ from $\mathcal{A}$ otherwise.
2. If $\mathsf{type}_2 =$ "open", for $i \in [1, m]$, $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{OLEe}}$ by receiving $r'_{\mathcal{P},i} \in \mathbb{F}_{2^\lambda}$, $a_i \in \mathbb{F}_{2^\lambda}$ and an error vector $\boldsymbol{e}_i \in (\mathbb{F}_{2^\lambda})^\lambda$ from $\mathcal{A}$, and then $\mathcal{S}$ sets an error $s_i := r'_{\mathcal{P},i} - (r_{\mathcal{P}})^i$ where $r_{\mathcal{P}} = r'_{\mathcal{P},1}$. Then, for $i \in [1, m]$, $\mathcal{S}_{\mathsf{AEAD}}$ defines an error $e_i' := (\boldsymbol{g} * \boldsymbol{e}_i) \odot \boldsymbol{v}_i + s_i \cdot (r_{\mathcal{V}})^i \in \mathbb{F}_{2^\lambda}$, where $r_{\mathcal{V}} \in \mathbb{F}_{2^\lambda}$ is sampled uniformly by $\mathcal{S}_{\mathsf{AEAD}}$ and $\boldsymbol{v}_i$ is the bit-decomposition of $(r_{\mathcal{V}})^i$. For $i \in [1, m]$, $\mathcal{S}$ computes $\mathcal{V}$'s share on $\left[r^i\right]_{2^{128}}$ as $b_i = (r_{\mathcal{P}} \cdot r_{\mathcal{V}})^i - a_i + e_i'$.
3. In the handshake/record phase, given an application key key' from the simulation of the main protocol Π_{AuthData}, $\mathcal{S}_{\mathsf{AEAD}}$ simulates as follows:
   - If $\mathsf{type}_2 =$ "open", then simulator $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving a circuit $f_{\mathsf{AES}}^{(1)}$ and a tuple $(h'_{\mathcal{P}}, \mathrm{z}'_{0,\mathcal{P}}, \mathsf{st}'_0, \mathsf{st}'_1)$. Then, $\mathcal{S}$ computes $(h'_{\mathcal{V}}, \mathrm{z}'_{0,\mathcal{V}}, \mathrm{z}'_1) \leftarrow f_{\mathsf{AES}}^{(1)}(\mathsf{key}', h'_{\mathcal{P}}, \mathrm{z}'_{0,\mathcal{P}}, \mathsf{st}'_0, \mathsf{st}'_1)$, and sends z'_1 to $\mathcal{A}$.
   - If $\mathsf{type}_2 =$ "secret", then simulator $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving a "malicious" circuit $f_{\mathsf{AES}}^{(2)}$ and a tuple $(\mathrm{z}'_{0,\mathcal{P}}, \ldots, \mathrm{z}'_{n,\mathcal{P}}, \mathsf{st}'_0, \ldots, \mathsf{st}'_n)$ from adversary $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes $(\mathrm{z}'_{0,\mathcal{V}}, \ldots, \mathrm{z}'_{n,\mathcal{V}}) \leftarrow f_{\mathsf{AES}}^{(2)}(\mathsf{key}', \mathrm{z}'_{0,\mathcal{P}}, \ldots, \mathrm{z}'_{n,\mathcal{P}}, \mathsf{st}'_0, \ldots, \mathsf{st}'_n)$.
4. In the handshake/record phase, $\mathcal{S}_{\mathsf{AEAD}}$ simulates the AEAD encryption/decryption as follows:
   - If $\mathsf{type}_1 =$ "encryption" and $\mathsf{type}_2 =$ "open", then given M_1, $\mathcal{S}_{\mathsf{AEAD}}$ computes $\mathbf{C} := \mathrm{z}_1 \oplus \mathrm{M}_1$.
   - If $\mathsf{type}_1 =$ "decryption" and $\mathsf{type}_2 =$ "open", then given C_1, $\mathcal{S}_{\mathsf{AEAD}}$ computes $\mathbf{M} := \mathrm{z}_1 \oplus \mathrm{C}_1$.
   - If $\mathsf{type}_1 =$ "encryption" and $\mathsf{type}_2 =$ "secret", then for $i \in [1, n]$, $\mathcal{S}_{\mathsf{AEAD}}$ receives B_i from $\mathcal{A}$, and extracts $\mathrm{M}_i := \mathrm{B}_i \oplus \mathrm{z}_{i,\mathcal{P}}$ and sets $\mathbf{M} := (\mathrm{M}_1, \ldots, \mathrm{M}_n)$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes $\mathrm{C}_i := \mathrm{B}_i \oplus \mathrm{z}'_{i,\mathcal{V}}$ for $i \in [1, n]$, sets $\mathbf{C} = (\mathrm{C}_1, \ldots, \mathrm{C}_n)$ and sends $\mathbf{C}$ to $\mathcal{A}$.
   - If $\mathsf{type}_1 =$ "decryption" and $\mathsf{type}_2 =$ "secret", then $\mathcal{S}_{\mathsf{AEAD}}$ sends $\mathrm{z}'_{i,\mathcal{V}}$ for all $i \in [1, n]$ to $\mathcal{A}$.
5. If $\mathsf{type}_2 =$ "open", $\mathcal{S}_{\mathsf{AEAD}}$ simulates the $\mathsf{Open}([h]_{2^{128}} - [r]_{2^{128}})$ procedure by sending $d_2' = h'_{\mathcal{V}} - r_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$ to $\mathcal{A}$ and receiving $d_1' \in \mathbb{F}_{2^{128}}$ from $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes an error $e' := d_1' - (h_{\mathcal{P}} - r_{\mathcal{P}}) \in \mathbb{F}_{2^{128}}$ and sets $d' := d_1' + d_2' \in \mathbb{F}_{2^{128}}$. For each $i \in [2, m]$, $\mathcal{S}_{\mathsf{AEAD}}$ also computes the shares of both parties on $\left[h^i\right]_{2^{128}}$ following the protocol specification using d' and a_j, b_j for all $j \in [1, i]$.
6. Following the protocol description, $\mathcal{S}_{\mathsf{AEAD}}$ computes the shares of $\mathcal{P}$ and $\mathcal{V}$ on $[\sigma]_{2^{128}}$ that are denoted by $\sigma_{\mathcal{P}}$ and $\sigma_{\mathcal{V}}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ simulates the generation of a GMAC tag as follows:
   - If $\mathsf{type}_1 =$ "encryption", $\mathcal{S}_{\mathsf{AEAD}}$ sends $\sigma_{\mathcal{V}}$ to $\mathcal{A}$, and receives $\sigma'_{\mathcal{P}}$ from $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes $\sigma := \sigma'_{\mathcal{P}} \oplus \sigma_{\mathcal{V}}$ and sets $\mathrm{CT} = (\mathbf{C}, \sigma)$.
   - If $\mathsf{type}_1 =$ "decryption", $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{Com}}$ by receiving $\sigma'_{\mathcal{P}}$ from $\mathcal{A}$ and then opening $\sigma_{\mathcal{V}}$ to $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes $\sigma := \sigma'_{\mathcal{P}} \oplus \sigma_{\mathcal{V}}$ and uses σ to check validity of an AEAD ciphertext CT given to $\mathcal{S}_{\mathsf{AEAD}}$. If the check fails, $\mathcal{S}_{\mathsf{AEAD}}$ aborts.

In both cases, $\mathcal{S}_{\mathsf{AEAD}}$ computes $e_\sigma := \sigma'_{\mathcal{P}} - \sigma_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes an error $E := L(e'_1, \dots, e'_m) + M(r, e') + e_\sigma$ where L is a linear function and M is a polynomial function that depends on the protocol specification.

7. In the post-record phase, given key^* from the simulation of the main protocol Π_{AuthData}, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (output) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as follows:
   - If $\mathsf{type}_2 =$ "open", $\mathcal{S}_{\mathsf{AEAD}}$ computes $h' := h_{\mathcal{P}} \oplus h'_{\mathcal{V}}$ and $\mathrm{z}'_0 := \mathrm{z}_{0,\mathcal{P}} \oplus \mathrm{z}'_{0,\mathcal{V}}$.
   - If $\mathsf{type}_2 =$ "secret", $\mathcal{S}_{\mathsf{AEAD}}$ computes $\mathrm{z}'_0 := \mathrm{z}_{0,\mathcal{P}} \oplus \mathrm{z}'_{0,\mathcal{V}}$.
8. Given st^*, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as follows:
   - If $\mathsf{type}_2 =$ "open", then $\mathcal{S}_{\mathsf{AEAD}}$ checks that $h' = \mathsf{AES}(\mathsf{key}^*, \mathbf{0})$, $\mathrm{z}'_0 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^*)$, and $\mathrm{z}'_1 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^* + 1)$.
   - If $\mathsf{type}_2 =$ "secret", then $\mathcal{S}_{\mathsf{AEAD}}$ checks that $\mathrm{z}'_0 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^*)$ and $\mathrm{z}'_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^* + i) \oplus \mathrm{z}_{i,\mathcal{P}}$ for all $i \in [1, n]$.

   If any check fails, $\mathcal{S}_{\mathsf{AEAD}}$ aborts.

Simulation and analysis of main protocol Π_{AuthData}. Given $\mathcal{S}_{\mathsf{E2F}}$, $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$, simulator $\mathcal{S}$ emulates the functionalities used in protocol Π_{AuthData}, and interacts with adversary $\mathcal{A}$ as follows.

1. $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{E2F}}$ to simulate the preprocessing phase of sub-protocol Π_{E2F} and $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the preprocessing phase of sub-protocol Π_{AEAD}. In the preprocessing phase, $\mathcal{S}$ records $\mathrm{z}_{i,\mathcal{P}}$ for all $i \in [1, n]$ that are sent by $\mathcal{A}$ to functionality $\mathcal{F}_{\mathsf{GP2PC}}$.
2. Following the TLS specification, on behalf of the server, $\mathcal{S}$ receives REQ_C from $\mathcal{A}$ and sends RES_S to $\mathcal{A}$. After receiving $(\mathrm{REQ}_C{}', \mathrm{RES}_S{}')$ from $\mathcal{A}$, $\mathcal{S}$ (on behalf of $\mathcal{V}$) checks the validity of $\mathrm{RES}_S{}'$ following the protocol description, and checks that $\mathrm{REQ}_C = \mathrm{REQ}_C{}'$ and $(r_S, T_S) = (r'_S, T'_S)$ where (r_S, T_S) and (r'_S, T'_S) are included in RES_S and $\mathrm{RES}_S{}'$ respectively. If any check fails, $\mathcal{S}$ aborts.
3. Following the protocol specification, $\mathcal{S}$ samples $t_{\mathcal{V}} \leftarrow \mathbb{Z}_q$ and computes $T_{\mathcal{V}} := t_{\mathcal{V}} \cdot G$, and then sends $T_{\mathcal{V}}$ to $\mathcal{A}$. On behalf of $\mathcal{V}$, $\mathcal{S}$ receives $\mathrm{RES}_C = T_C$ from $\mathcal{A}$; On behalf of the server, $\mathcal{S}$ receives $\mathrm{RES}_C{}' = T'_C$ from $\mathcal{A}$.
4. $\mathcal{S}$ computes $Z_2 := t_{\mathcal{V}} \cdot T_S$. Then, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{E2F}}$ to simulate the handshake phase of sub-protocol Π_{E2F}. With Z_2, $\mathcal{S}$ also computes $\widetilde{\mathsf{pms}}_{\mathcal{V}}$ following the specification of sub-protocol Π_{E2F}.
5. $\mathcal{S}$ defines $\mathsf{pms}_{\mathcal{V}}$ as the bit decomposition of $\widetilde{\mathsf{pms}}_{\mathcal{V}} \in \mathbb{Z}_p$. Then, $\mathcal{S}$ emulates the (eval) command of $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving $\mathsf{pms}'_{\mathcal{P}} \in \{0,1\}^{\lceil \log p \rceil}$ and a circuit f_{ADD} from $\mathcal{A}$. $\mathcal{S}$ stores $\mathsf{pms}'_{\mathcal{P}}$ and computes $\mathsf{pms}' := f_{\mathsf{ADD}}(\mathsf{pms}_{\mathcal{V}})$.
6. Given pms' and $r_C \| r_S$ where r_C is included in REQ_C and r_S is involved in RES_S, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ towards $\mathsf{type} =$ "secret" to simulate the handshake phase of sub-protocol execution $\Pi^{(1)}_{\mathsf{PRF}}$ and obtains a master secret ms'.
7. Given ms' and $r_S \| r_C$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ towards $\mathsf{type} =$ "partial open" to simulate the handshake phase of sub-protocol execution $\Pi^{(2)}_{\mathsf{PRF}}$. During this execution, $\mathcal{S}$ obtains application keys $\mathsf{key}'_C, \mathsf{key}'_S$ and sends IV'_C, IV'_S to $\mathcal{A}$. Then, $\mathcal{S}$ initializes $(\mathsf{st}^C_e, \mathsf{st}^C_d) := (IV'_C, IV'_S)$.
8. $\mathcal{S}$ computes $\tau_C := \mathsf{H}(\mathrm{REQ}_C \| \mathrm{RES}_S{}' \| \mathrm{RES}_C)$. Given ms' and τ_C, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ towards $\mathsf{type} =$ "open" to simulate the handshake phase of sub-protocol execution $\Pi^{(3)}_{\mathsf{PRF}}$. During this execution, $\mathcal{S}$ sends UFIN_C to $\mathcal{A}$.
9. Given $(\mathsf{key}'_C, \mathsf{st}^C_e, \ell_C, \mathrm{H}_C, \mathrm{UFIN}_C)$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the sub-protocol execution $\Pi^{(1)}_{\mathsf{AEAD}}$ towards $\mathsf{type}_1 =$ "encryption" and $\mathsf{type}_2 =$ "open". During this execution, $\mathcal{S}$ obtains FIN_C and

$\mathcal{P}$'s shares on $\left[h_C^i\right]_{2^{128}}$ for $i \in [1,m]$. On behalf of the server, $\mathcal{S}$ receives $(\textsc{h}'_C, \textsc{fin}_C{}')$ from $\mathcal{A}$ and checks $\textsc{fin}_C{}'$ following the TLS specification. If the check fails, $\mathcal{S}$ aborts. Then, $\mathcal{S}$ updates $\mathsf{st}_e^C := \mathsf{st}_e^C + 2$.

10. $\mathcal{S}$ computes $\tau_S := \mathsf{H}(\textsc{req}_C \| \textsc{res}_S{}' \| \textsc{res}_C \| \textsc{ufin}_C)$. Given ms' and τ_S, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ towards type = "open" to simulate the handshake phase of sub-protocol execution $\Pi_{\mathsf{PRF}}^{(4)}$. During this execution, $\mathcal{S}$ sends $\textsc{ufin}_S$ to $\mathcal{A}$.
11. On behalf of the server, $\mathcal{S}$ generates $(\textsc{h}_S, \textsc{fin}_S)$ following the TLS specification and sends it to $\mathcal{A}$. On behalf of $\mathcal{V}$, simulator $\mathcal{S}$ receives $(\textsc{h}'_S, \textsc{fin}_S{}')$ from $\mathcal{A}$. Given $(\mathsf{key}'_S, \mathsf{st}_d^C, \ell_S, \textsc{h}'_S, \textsc{fin}_S{}')$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the sub-protocol execution $\Pi_{\mathsf{AEAD}}^{(2)}$ towards type_1 = "decryption" and type_2 = "open". During this execution, $\mathcal{S}$ checks $\textsc{fin}_S{}'$ on behalf of $\mathcal{V}$ and aborts if the check fails. $\mathcal{S}$ also gets $\mathcal{P}$'s shares on $\left[h_S^i\right]_{2^{128}}$ for $i \in [1,m]$. Then, $\mathcal{S}$ updates $\mathsf{st}_d^C := \mathsf{st}_d^C + 2$.
12. Given $(\mathsf{key}'_C, \mathsf{st}_e^C, \ell_Q, \textsc{h}_Q)$ and $\mathcal{P}$'s shares on $\left[h_C^i\right]_{2^{128}}$ for $i \in [1,m]$, simulator $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the sub-protocol execution $\Pi_{\mathsf{AEAD}}^{(3)}$ towards type_1 = "encryption" and type_2 = "secret". During this execution, $\mathcal{S}$ extracts a query Q, computes an AEAD ciphertext $\textsc{enc}_Q$, and records $\textsc{z}_{i,\mathcal{V}}$ with $\textsc{z}_{i,\mathcal{P}} \oplus \textsc{z}_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}_C, \mathsf{st}_e^C + i)$ for $i \in [1,n]$. Then, $\mathcal{S}$ sends Q to functionality $\mathcal{F}_{\mathsf{AuthData}}$ and receives a response R from $\mathcal{F}_{\mathsf{AuthData}}$.
13. On behalf of the server, $\mathcal{S}$ receives $(\textsc{h}'_Q, \textsc{enc}_Q{}')$ from $\mathcal{A}$, and then checks correctness of the pair following the TLS specification and aborts if the check fails. On behalf of the server, $\mathcal{S}$ generates $(\textsc{h}_R, \textsc{enc}_R)$ with R following the TLS specification, and sends it to $\mathcal{A}$. On behalf of honest verifier $\mathcal{V}$, $\mathcal{S}$ receives $(\textsc{h}'_R, \textsc{enc}_R{}')$ from $\mathcal{A}$.
14. $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the post-record phase of sub-protocol executions $\Pi_{\mathsf{AEAD}}^{(1)}$, $\Pi_{\mathsf{AEAD}}^{(2)}$ and $\Pi_{\mathsf{AEAD}}^{(3)}$. During this execution, $\mathcal{S}$ obtains $(h_C, h_S, \textsc{z}_C, \textsc{z}_S, \textsc{z}_Q)$.
15. $\mathcal{S}$ emulates the $(\mathsf{revealandprove})$ command of $\mathcal{F}_{\mathsf{GP2PC}}$ by sending $\mathsf{pms}_{\mathcal{V}}$ to $\mathcal{A}$. In parallel, $\mathcal{S}$ sends $t_{\mathcal{V}}$ to $\mathcal{A}$.
16. $\mathcal{S}$ emulates the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on $\mathsf{pms}'_{\mathcal{P}}$ stored previously and circuit $\mathit{AddModp}$ to compute $\mathsf{pms}^* = \mathit{AddModp}(\mathsf{pms}'_{\mathcal{P}}, \mathsf{pms}_{\mathcal{V}})$.
17. Given pms^*, simulator $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$ for the post-record phase of sub-protocol executions $\Pi_{\mathsf{PRF}}^{(1)}$, $\Pi_{\mathsf{PRF}}^{(2)}$, $\Pi_{\mathsf{PRF}}^{(3)}$, $\Pi_{\mathsf{PRF}}^{(4)}$, $\Pi_{\mathsf{AEAD}}^{(1)}$, $\Pi_{\mathsf{AEAD}}^{(2)}$ and $\Pi_{\mathsf{AEAD}}^{(3)}$ to check the correctness of all values obtained by $\mathcal{V}$. If the check fails, then $\mathcal{S}$ aborts. During these executions, $\mathcal{S}$ computes
$$\mathsf{ms}^* = \mathsf{PRF}_{384}(\mathsf{pms}^*, \text{“master secret”}, r_C \| r_S),$$
$$(\mathsf{key}_C^*, IV_C{}^*, \mathsf{key}_S^*, IV_S{}^*) = \mathsf{PRF}_{320}(\mathsf{ms}^*, \text{“key expansion”}, r_S \| r_C).$$
18. $\mathcal{S}$ emulates the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by computing $\textsc{z}_R := \mathsf{AES}(\mathsf{key}_S^*, \mathsf{st}_d^C)$ and receiving MAC tags on $[\![\textsc{z}_R]\!]$ from $\mathcal{A}$. After receiving $\textsc{z}'_R$ from $\mathcal{A}$, simulator $\mathcal{S}$ emulates the (check) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on $[\![\textsc{z}_R]\!] - \textsc{z}'_R$ by sending true to $\mathcal{A}$ if $\textsc{z}'_R = \textsc{z}_R$ or false to $\mathcal{A}$ otherwise. Following the protocol specification, $\mathcal{S}$ uses $(h_C, \textsc{z}_C, \textsc{z}_Q)$ to check correctness of all GMAC tags in AEAD ciphertexts $\textsc{fin}_C, \textsc{enc}_Q$, and aborts if the check fails. If $\textsc{fin}_S{}' \neq \textsc{fin}_S$ or $\textsc{enc}_R{}' \neq \textsc{enc}_R$, then $\mathcal{S}$ aborts.
19. $\mathcal{S}$ emulates the $(\mathsf{authinput})$ command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving MAC tags on $[\![\textsc{z}_{i,\mathcal{P}}]\!]$ for $i \in [1,n]$ from $\mathcal{A}$. Then, $\mathcal{S}$ computes MAC tags on $[\![\textsc{z}_i]\!] = [\![\textsc{z}_{i,\mathcal{P}}]\!] \oplus \textsc{z}_{i,\mathcal{V}}$ for $i \in [1,n]$. Next, $\mathcal{S}$ computes MAC tags on $[\![\textsc{m}_i]\!] := [\![\textsc{z}_i]\!] \oplus \textsc{c}_i$ for $i \in [1,n]$ where $\mathbf{C}_Q = (\textsc{c}_1, \ldots, \textsc{c}_n)$ is the AES ciphertext included in $\textsc{enc}_Q$ and $[\![Q]\!] = ([\![\textsc{m}_1]\!], \ldots, [\![\textsc{m}_n]\!])$.
20. $\mathcal{S}$ emulates the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving MAC tags on $[\![R]\!]$.

21. $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{Conv}}$ to convert $(\llbracket Q \rrbracket, \llbracket R \rrbracket)$ into additively homomorphic commitments by sending commitment identifiers to $\mathcal{A}$.

Malicious verifier $\mathcal{V}$. We first show the constructions of $\mathcal{S}_{\mathsf{E2F}}$, $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$ for three sub-protocols, and then give the construction of $\mathcal{S}$ for main protocol Π_{AuthData}. As such, these simulators emulate the same functionalities $\mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{IZK}}, \mathcal{F}_{\mathsf{Conv}}$, and only simulator $\mathcal{S}$ is given access to functionality $\mathcal{F}_{\mathsf{AuthData}}$.

Simulation and analysis of sub-protocol Π_{E2F}. $\mathcal{S}_{\mathsf{E2F}}$ emulates functionality $\mathcal{F}_{\mathsf{OLEe}}$, and interacts with adversary $\mathcal{A}$ as follows.

1. $\mathcal{S}_{\mathsf{E2F}}$ emulates functionality $\mathcal{F}_{\mathsf{OLEe}}$ by receiving $(b_2, a_2, b'_2, a_2, r_2)$ and $\mathcal{V}$'s shares on $[a_1 b_2]_p$, $[a_2 b_1]_p$, $[a_1 b'_2]_p$, $[a_2 b'_1]_p$ and $[r_1 r_2]_p$.
2. $\mathcal{S}_{\mathsf{E2F}}$ computes the shares of $[c]_p$, $[c']_p$ and $[r^2]_p$ held by $\mathcal{V}$ following the protocol specification.
3. On behalf of honest $\mathcal{P}$, $\mathcal{S}_{\mathsf{E2F}}$ sends random field elements in $\mathbb{Z}_p$ to simulate the Open procedure on the values $\epsilon_1, \epsilon_2, \epsilon_3, w$ opened.
4. During the Open procedure on the values $\epsilon_1, \epsilon_2, \epsilon_3, w$, $\mathcal{S}_{\mathsf{E2F}}$ extracts the errors e_6, e_7, e_8, e_9 in the way similar to the case of a malicious prover.
5. $\mathcal{S}_{\mathsf{E2F}}$ computes an error $e_z = g(a, r, e_6, e_7, e_8, e_9)$, which is added into the secret on $[z]_p$, where g is a function depending on the definition of z.

As such, we put sub-protocol Π_{E2F} into the main protocol Π_{AuthData} to analyze the security in the case of a malicious verifier $\mathcal{V}$. Similar to the analysis of simulation by $\mathcal{S}_{\mathsf{E2F}}$ for the case of malicious prover, based on the high entropy of a, r, we obtain that all the errors e_6, e_7, e_8, e_9 have to be zero, if the main protocol execution Π_{AuthData} does not abort. In this case, adversary $\mathcal{A}$ cannot leak any secret information on the EC point $Z_1 = (x_1, y_1)$.

Simulation and analysis of sub-protocol Π_{PRF}. $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$, and interacts with adversary $\mathcal{A}$.

1. $\mathcal{S}_{\mathsf{PRF}}$ is given secret' from the simulation of the main protocol Π_{AuthData} in the handshake phase. $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $IV_2 = f_{\mathsf{H}}(IV_0, \mathsf{secret}' \oplus \text{opad})$, receiving an error e_1 from $\mathcal{A}$ and sending $IV_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret}' \oplus \text{ipad})$ to $\mathcal{A}$. Then $\mathcal{S}_{\mathsf{PRF}}$ sets $IV'_1 := IV_1 \oplus e_1$ that is implicitly opened to honest prover $\mathcal{P}$.
2. From $i = 1$ to n, $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving an error $e_{2,i}$ from adversary $\mathcal{A}$, setting $M'_i = f_{\mathsf{H}}(IV_2, W'_i) \oplus e_{2,i}$ and sending $M_i = f_{\mathsf{H}}(IV_2, W'_i)$ to $\mathcal{A}$ where $W'_i = f_{\mathsf{H}}(IV'_1, M'_{i-1})$ and $M'_0 = \mathit{label} \| \mathit{msg}$. Here M'_i is implicitly opened to honest prover $\mathcal{P}$.
3. $\mathcal{S}_{\mathsf{PRF}}$ computes $X'_i := f_{\mathsf{H}}(IV'_1, M'_i \| \mathit{label} \| \mathit{msg})$ for each $i \in [1, n]$. Then, $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by computing an output $\mathsf{der} := \big(f_{\mathsf{H}}(IV_2, X'_1), \ldots, f_{\mathsf{H}}(IV_2, X'_{n-1}), \mathsf{Trunc}_m(f_{\mathsf{H}}(IV_2, X'_n))\big)$.
4. If $\mathsf{type} =$ "open", $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving an error e_3 from $\mathcal{A}$ and sending der to $\mathcal{A}$. In this case, $\mathcal{S}_{\mathsf{PRF}}$ also computes $\mathsf{der}' = \mathsf{der} \oplus e_3$ that is implicitly opened to honest prover $\mathcal{P}$. If $\mathsf{type} =$ "partial open", $\mathcal{S}_{\mathsf{PRF}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by parsing $\mathsf{der} = (\mathsf{key}_C, IV_C, \mathsf{key}_S, IV_S)$, sending (IV_C, IV_S) to $\mathcal{A}$ and receiving an error (e_4, e_5) from $\mathcal{A}$. In this case, $\mathcal{S}_{\mathsf{PRF}}$ also computes $IV'_C = IV_C \oplus e_4$ and $IV'_S = IV_S \oplus e_5$ that are implicitly opened to honest $\mathcal{P}$.
5. In the post-record phase, simulator $\mathcal{S}_{\mathsf{PRF}}$ is given secret^* from the simulation of the main protocol Π_{AuthData}, where secret^* is a correct secret. In the simulation of main protocol Π_{AuthData}, the simulator computes $IV_2^* = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \text{opad})$, and checks the following values that are computed correctly with secret^* and IV_2^*.

$$IV'_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \text{ipad});$$

$M'_i = f_{\mathsf{H}}(IV_2^*, W'_i)$ where $W'_i = f_{\mathsf{H}}(IV'_1, M'_{i-1})$;
If type="open", $\mathsf{der}' = \big(f_{\mathsf{H}}(IV_2^*, X'_1), \ldots, f_{\mathsf{H}}(IV_2^*, X'_{n-1}),$
$\mathsf{Trunc}_m(f_{\mathsf{H}}(IV_2^*, X'_n))\big)$ where $X'_i := f_{\mathsf{H}}(IV'_1, M'_i \| label \| msg)$;
If type="partial open", $IV'_C = IV^*_C$ and $IV'_S = IV^*_S$, where
IV^*_C, IV^*_S are computed with $IV_2^*, X'_1, \ldots, X'_n$.

If any check fails, the simulator of protocol Π_{AuthData} aborts. In this case, $\mathcal{S}_{\mathsf{PRF}}$ aborts.

6. $\mathcal{S}_{\mathsf{PRF}}$ emulates the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by always sending true to $\mathcal{A}$.

Similarly, we put sub-protocol Π_{PRF} into the main protocol Π_{AuthData} for analyzing the security in the case of a malicious verifier $\mathcal{V}$. Clearly, the simulation for the (eval) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ is perfect. If the main-protocol execution does not abort, then $\mathsf{secret}' = \mathsf{secret}^*$ based on the analysis in the proof of Theorem 1. Furthermore, in this case, the values opened are correctly computed with secret^*, i.e., any errors introduced by $\mathcal{A}$ to these values would incur abort. In the real protocol execution, if honest $\mathcal{P}$ does not abort, then functionality $\mathcal{F}_{\mathsf{GP2PC}}$ always outputs true to $\mathcal{V}$. Therefore, the simulation for the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ is also perfect. As in the analysis for the case that $\mathcal{P}$ is malicious, $\mathcal{A}$ cannot learn secret^* for all three cases of type, der^* if type = "secret" and $(\mathsf{key}^*_C, \mathsf{key}^*_S)$ if type = "partial open" from the values opened, as f_{H} is a random oracle, and both of secret^* and IV_2^* have sufficiently high entropy.

Simulation and analysis of sub-protocol Π_{AEAD}. We construct the simulator $\mathcal{S}_{\mathsf{AEAD}}$ for all four cases, although the case of type_1 = "decryption" and type_2 = "secret" is *not* necessary for the simulation of main protocol Π_{AuthData}. Simulator $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionalities $\mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{Com}}$, and interacts with adversary $\mathcal{A}$ as follows.

1. In the preprocessing phase, if type_2 = "open", then for $i \in [1, m]$, $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{OLEe}}$ by receiving $r'_{\mathcal{V},i} \in \mathbb{F}_{2^\lambda}$ and $b_i \in \mathbb{F}_{2^\lambda}$ from $\mathcal{A}$. Then, for $i \in [1, m]$, $\mathcal{S}$ sets an error $e_i := r'_{\mathcal{V},i} - (r_{\mathcal{V}})^i$ where $r_{\mathcal{V}} = r'_{\mathcal{V},1}$. Then, for $i \in [1, m]$, $\mathcal{S}_{\mathsf{AEAD}}$ defines an error $e'_i := e_i \cdot (r_{\mathcal{P}})^i \in \mathbb{F}_{2^\lambda}$, where $r_{\mathcal{P}} \in \mathbb{F}_{2^\lambda}$ is sampled at random by $\mathcal{S}_{\mathsf{AEAD}}$. For $i \in [1, m]$, $\mathcal{S}$ computes $\mathcal{P}$'s share on $\left[r^i\right]_{2^{128}}$ as $a_i = (r_{\mathcal{P}} \cdot r_{\mathcal{V}})^i - b_i + e'_i$.
2. In the handshake/record phase, given an application key key' and st' from the simulation of the main protocol Π_{AuthData}, $\mathcal{S}_{\mathsf{AEAD}}$ simulates as follows:
   - If type_2 = "open", then $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $(h_{\mathcal{V}}, \mathrm{z}_{0,\mathcal{V}}, \mathrm{z}_1) \leftarrow \mathcal{C}_{aes}(\mathsf{key}', h_{\mathcal{P}}, \mathrm{z}_{0,\mathcal{P}}, \mathsf{st}', \mathsf{st}'+1)$ and sending $h_{\mathcal{V}}, \mathrm{z}_{0,\mathcal{V}}, \mathrm{z}_1$ to $\mathcal{A}$, where $h_{\mathcal{P}}, \mathrm{z}_{0,\mathcal{P}} \in \mathbb{F}_{2^{128}}$ are sampled uniformly by $\mathcal{S}_{\mathsf{AEAD}}$. Besides, $\mathcal{S}_{\mathsf{AEAD}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving an error $e_z \in \mathbb{F}_{2^{128}}$ from $\mathcal{A}$ and setting $\mathrm{z}'_1 = \mathrm{z}_1 + e_z \in \mathbb{F}_{2^{128}}$.
   - If type_2 = "secret", then $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $(\mathrm{z}_{0,\mathcal{V}}, \ldots, \mathrm{z}_{n,\mathcal{V}}) \leftarrow \mathcal{D}_{aes}(\mathsf{key}', \mathrm{z}_{0,\mathcal{P}}, \ldots, \mathrm{z}_{n,\mathcal{P}}, \mathsf{st}', \ldots, \mathsf{st}'+n)$ and sending $\mathrm{z}_{i,\mathcal{V}}$ for $i \in [0, n]$ to $\mathcal{A}$, where $\mathrm{z}_{0,\mathcal{P}}, \ldots, \mathrm{z}_{n,\mathcal{P}} \in \mathbb{F}_{2^{128}}$ are sampled at random by $\mathcal{S}_{\mathsf{AEAD}}$.
3. In the handshake/record phase, depending on type_1 and type_2, $\mathcal{S}_{\mathsf{AEAD}}$ simulates the AEAD encryption/decryption as follows:
   - If type_1 = "encryption" and type_2 = "open", then given M_1, $\mathcal{S}_{\mathsf{AEAD}}$ computes $\mathbf{C} := \mathrm{z}'_1 \oplus \mathrm{M}_1$.
   - If type_1 = "decryption" and type_2 = "open", then given C_1, $\mathcal{S}_{\mathsf{AEAD}}$ computes $\mathbf{M} := \mathrm{z}'_1 \oplus \mathrm{C}_1$.
   - If type_1 = "encryption" and type_2 = "secret", then for $i \in [1, n]$, $\mathcal{S}_{\mathsf{AEAD}}$ samples $\mathrm{B}_i \leftarrow \mathbb{F}_{2^{128}}$ and sends B_i to $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ receives $\mathbf{C}' = (\mathrm{C}'_1, \ldots, \mathrm{C}'_n)$ from $\mathcal{A}$.
   - If type_1 = "decryption" and type_2 = "secret", then given $\mathbf{C} = (\mathrm{C}_1, \ldots, \mathrm{C}_n)$, $\mathcal{S}_{\mathsf{AEAD}}$ receives $\mathrm{z}'_{i,\mathcal{V}}$ for all $i \in [1, n]$ from $\mathcal{A}$.

4. If $\mathsf{type}_2 =$ “open”, $\mathcal{S}_{\mathsf{AEAD}}$ simulates the $\mathsf{Open}([h]_{2^{128}} - [r]_{2^{128}})$ procedure by sending $d_1 = h_{\mathcal{P}} - r_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$ to $\mathcal{A}$ and receiving $d_2' \in \mathbb{F}_{2^{128}}$ from $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes an error $e' := d_2' - (h_{\mathcal{V}} - r_{\mathcal{V}}) \in \mathbb{F}_{2^{128}}$ and sets $d' := d_1 + d_2' \in \mathbb{F}_{2^{128}}$. For $i \in [2, m]$, $\mathcal{S}_{\mathsf{AEAD}}$ computes the shares of both parties on $\left[h^i\right]_{2^{128}}$ following the protocol specification using d' and a_j, b_j for all $j \in [1, i]$.

5. Following the protocol specification, $\mathcal{S}_{\mathsf{AEAD}}$ computes the shares of $\mathcal{P}$ and $\mathcal{V}$ on $[\sigma]_{2^{128}}$ that are denoted by $\sigma_{\mathcal{P}}$ and $\sigma_{\mathcal{V}}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ simulates the generation of a GMAC tag as follows:

   - If $\mathsf{type}_1 =$ “encryption”, $\mathcal{S}_{\mathsf{AEAD}}$ sends $\sigma_{\mathcal{P}}$ to $\mathcal{A}$, and receives $\sigma_{\mathcal{V}}'$ from $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes $\sigma := \sigma_{\mathcal{P}} \oplus \sigma_{\mathcal{V}}'$ and sets $\textsc{ct} = (\mathbf{C}, \sigma)$.
   - If $\mathsf{type}_1 =$ “decryption”, $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{Com}}$ by receiving $\sigma_{\mathcal{V}}'$ from $\mathcal{A}$ and then opening $\sigma_{\mathcal{P}}$ to $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes $\sigma := \sigma_{\mathcal{P}} \oplus \sigma_{\mathcal{V}}'$ and uses σ to check validity of an AEAD ciphertext $\textsc{ct}$ given to $\mathcal{S}_{\mathsf{AEAD}}$. If the check fails, $\mathcal{S}_{\mathsf{AEAD}}$ aborts.

   In both cases, $\mathcal{S}_{\mathsf{AEAD}}$ computes $e_\sigma := \sigma_{\mathcal{V}}' - \sigma_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes an error $E := L(e_1', \ldots, e_m') + M(r, e') + e_\sigma$.

6. In the post-record phase, given key^* from the simulation of the main protocol Π_{AuthData}, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (output) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as follows:

   - If $\mathsf{type}_2 =$ “open”, then $\mathcal{S}_{\mathsf{AEAD}}$ sends $(h_{\mathcal{P}}, \mathrm{z}_{0,\mathcal{P}})$ to $\mathcal{A}$, and computes $h := h_{\mathcal{P}} \oplus h_{\mathcal{V}}$ and $\mathrm{z}_0 := \mathrm{z}_{0,\mathcal{P}} \oplus \mathrm{z}_{0,\mathcal{V}}$.
   - If $\mathsf{type}_2 =$ “secret”, then $\mathcal{S}_{\mathsf{AEAD}}$ sends $\mathrm{z}_{0,\mathcal{P}}$ to $\mathcal{A}$, and computes $\mathrm{z}_0 := \mathrm{z}_{0,\mathcal{P}} \oplus \mathrm{z}_{0,\mathcal{V}}$.

7. Given st^*, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as follows:

   - If $\mathsf{type}_2 =$ “open”, then $\mathcal{S}_{\mathsf{AEAD}}$ checks that $h = \mathsf{AES}(\mathsf{key}^*, \mathbf{0})$, $\mathrm{z}_0 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^*)$, and $\mathrm{z}_1 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^* + 1)$.
   - If $\mathsf{type}_2 =$ “secret”, then $\mathcal{S}_{\mathsf{AEAD}}$ checks that $\mathrm{z}_0 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^*)$ and $\mathrm{z}_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^* + i) \oplus \mathrm{z}_{i,\mathcal{P}}$ for all $i \in [1, n]$.

   If the check fails, $\mathcal{S}_{\mathsf{AEAD}}$ sends false to $\mathcal{A}$ and then aborts. Otherwise, $\mathcal{S}_{\mathsf{AEAD}}$ sends true to $\mathcal{A}$.

Simulation and analysis of main protocol Π_{AuthData}. By invoking $\mathcal{S}_{\mathsf{E2F}}$, $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$, $\mathcal{S}$ emulates the functionalities used in protocol Π_{AuthData}, and interacts with $\mathcal{A}$ as follows.

1. $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{E2F}}$ to simulate the preprocessing phase of sub-protocol Π_{E2F} and $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the preprocessing phase of sub-protocol Π_{AEAD}.

2. Following the protocol description, $\mathcal{S}$ sends $(\textsc{req}_C, \textsc{res}_S)$ to $\mathcal{A}$, where $\mathcal{S}$ records $t_S \in \mathbb{Z}_q$ for $T_S = t_S \cdot G$ contained in $\textsc{res}_S$. After receiving $T_{\mathcal{V}}$, $\mathcal{S}$ samples $t_{\mathcal{P}} \leftarrow \mathbb{Z}_q$, computes $T_{\mathcal{P}} := t_{\mathcal{P}} \cdot G$ and sends $\textsc{res}_C = T_C = T_{\mathcal{P}} + T_{\mathcal{V}}$ to $\mathcal{A}$.

3. $\mathcal{S}$ computes $Z_1 := t_{\mathcal{P}} \cdot T_S$. Then, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{E2F}}$ to simulate the handshake phase of sub-protocol Π_{E2F}. With Z_1, simulator $\mathcal{S}$ computes $\widetilde{\mathsf{pms}}_{\mathcal{P}} \in \mathbb{Z}_p$ following the specification of sub-protocol $\mathcal{S}_{\mathsf{E2F}}$.

4. $\mathcal{S}$ emulates the (eval) command of $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving $\mathsf{pms}_{\mathcal{V}}' \in \{0,1\}^{\lceil \log p \rceil}$ from $\mathcal{A}$. Then, $\mathcal{S}$ sets $\mathsf{pms}_{\mathcal{P}}$ as the bit-decomposition of $\widetilde{\mathsf{pms}}_{\mathcal{P}} \in \mathbb{Z}_p$, and computes $\mathsf{pms}' := \mathit{AddModp}(\mathsf{pms}_{\mathcal{P}}, \mathsf{pms}_{\mathcal{V}}')$.

5. Given pms' and $r_C \| r_S$ where r_C is included in $\textsc{req}_C$ and r_S is involved in $\textsc{res}_S$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ towards $\mathsf{type} =$ “secret” to simulate the handshake phase of sub-protocol execution $\Pi_{\mathsf{PRF}}^{(1)}$ and obtains a master secret ms'.

6. Given ms' and $r_S \| r_C$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ towards $\mathsf{type} =$ “partial open” to simulate the handshake phase of sub-protocol execution $\Pi_{\mathsf{PRF}}^{(2)}$. During this execution, $\mathcal{S}$ obtains $\mathsf{key}_C', \mathsf{key}_S'$, sends

IV_C, IV_S to $\mathcal{A}$, and receives two errors e_1, e_2 from $\mathcal{A}$ when emulating functionality $\mathcal{F}_{\mathsf{GP2PC}}$. Then, $\mathcal{S}$ initializes $(\mathsf{st}_e^C, \mathsf{st}_d^C) = (IV_C', IV_S') = (IV_C \oplus e_1, IV_S \oplus e_2)$.

7. $\mathcal{S}$ computes $\tau_C := \mathsf{H}(\textsc{req}_C \| \textsc{res}_S \| \textsc{res}_C)$. Given ms' and τ_C, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ towards type = "open" to simulate the handshake phase of sub-protocol execution $\Pi_{\mathsf{PRF}}^{(3)}$. During this execution, by emulating functionality $\mathcal{F}_{\mathsf{GP2PC}}$, $\mathcal{S}$ sends $\textsc{ufin}_C$ to $\mathcal{A}$, receives an error e_3 from $\mathcal{A}$ and sets $\textsc{ufin}_C{}' := \textsc{ufin}_C \oplus e_3$.
8. Given $(\mathsf{key}_C', \mathsf{st}_e^C, \ell_C, \textsc{h}_C, \textsc{ufin}_C{}')$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the sub-protocol execution $\Pi_{\mathsf{AEAD}}^{(1)}$ towards type_1 = "encryption" and type_2 = "open". During this execution, $\mathcal{S}$ obtains $\textsc{fin}_C{}'$ and $\mathcal{V}$'s shares on $[h_C^i]_{2^{128}}$ for $i \in [1, m]$. On behalf of the server, $\mathcal{S}$ checks $(\textsc{h}_C, \textsc{fin}_C{}')$ following the TLS specification, and $\mathcal{S}$ aborts if the check fails. Then, $\mathcal{S}$ updates $\mathsf{st}_e^C := \mathsf{st}_e^C + 2$.
9. $\mathcal{S}$ computes $\tau_S := \mathsf{H}(\textsc{req}_C \| \textsc{res}_S \| \textsc{res}_C \| \textsc{ufin}_C{}')$. Given ms' and τ_S, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ towards type = "open" to simulate the handshake phase of sub-protocol execution $\Pi_{\mathsf{PRF}}^{(4)}$. During this execution, by emulating functionality $\mathcal{F}_{\mathsf{GP2PC}}$, $\mathcal{S}$ sends $\textsc{ufin}_S$ to $\mathcal{A}$, receives an error e_4 from $\mathcal{A}$ and sets $\textsc{ufin}_S{}' := \textsc{ufin}_S \oplus e_4$.
10. On behalf of the server, $\mathcal{S}$ generates $(\textsc{h}_S, \textsc{fin}_S)$ following the TLS specification, and then on behalf of $\mathcal{P}$, sends $(\textsc{h}_S, \textsc{fin}_S)$ to $\mathcal{A}$. Given $(\mathsf{key}_S', \mathsf{st}_d^C, \ell_S, \textsc{h}_S, \textsc{fin}_S)$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the sub-protocol execution $\Pi_{\mathsf{AEAD}}^{(2)}$ towards type_1 = "decryption" and type_2 = "open". During this execution, $\mathcal{S}$ checks $\textsc{fin}_S$ on behalf of $\mathcal{P}$ and aborts if the check fails. $\mathcal{S}$ also obtains $\mathcal{V}$'s shares on $[h_S^i]_{2^{128}}$ for $i \in [1, m]$. Then, $\mathcal{S}$ updates $\mathsf{st}_d^C := \mathsf{st}_d^C + 2$.
11. Given $(\mathsf{key}_C', \mathsf{st}_e^C, \ell_Q, \textsc{h}_Q)$ and $\mathcal{V}$'s shares on $[h_C^i]_{2^{128}}$ for $i \in [1, m]$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the sub-protocol execution $\Pi_{\mathsf{AEAD}}^{(3)}$ towards type_1 = "encryption" and type_2 = "secret". During this execution, $\mathcal{S}$ samples $\mathbf{C}_Q$ uniformly without knowing Q, and receives an AEAD ciphertext $\widetilde{\textsc{enc}_Q}$ from $\mathcal{A}$. For $i \in [1, n]$, $\mathcal{S}$ sends $\textsc{z}_{i,\mathcal{V}}$ to $\mathcal{A}$ where $\textsc{z}_{i,\mathcal{P}} \oplus \textsc{z}_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}_C, \mathsf{st}_e^C + i)$.
12. On behalf of the server, $\mathcal{S}$ receives $(\textsc{h}_Q', \textsc{enc}_Q{}')$ from $\mathcal{A}$. Then, $\mathcal{S}$ parses $\textsc{enc}_Q{}' = (\mathbf{C}_Q', \sigma_Q')$, and aborts if $\mathbf{C}_Q' \neq \mathbf{C}_Q$. Besides, on behalf of the server, $\mathcal{S}$ checks σ_Q' following the TLS specification. On behalf of the server, $\mathcal{S}$ samples $\mathbf{C}_R$ uniformly at random without knowing R, and then generates GMAC tag σ_R with key_S following the AEAD specification, where key_S is computed by $\mathcal{S}$ via simulating the server honestly following the TLS specification. Then, $\mathcal{S}$ sets $\textsc{enc}_R = (\mathbf{C}_R, \sigma_R)$ and sends $(\textsc{h}_R, \textsc{enc}_R)$ to $\mathcal{A}$. On behalf of honest prover $\mathcal{P}$, $\mathcal{S}$ receives $(\textsc{h}_R', \textsc{enc}_R{}')$ from $\mathcal{A}$.
13. $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the post-record phase of sub-protocol executions $\Pi_{\mathsf{AEAD}}^{(1)}$, $\Pi_{\mathsf{AEAD}}^{(2)}$ and $\Pi_{\mathsf{AEAD}}^{(3)}$. During this execution, $\mathcal{S}$ makes $\mathcal{A}$ obtain $(h_C, h_S, \textsc{z}_C, \textsc{z}_S, \textsc{z}_Q)$.
14. In the post-record phase, $\mathcal{S}$ receives $t_{\mathcal{V}} \in \mathbb{Z}_q$ from $\mathcal{A}$ and checks that $T_{\mathcal{V}} = t_{\mathcal{V}} \cdot G$. $\mathcal{S}$ also checks correctness of $\mathsf{pms}_{\mathcal{V}}'$ stored previously following the protocol description. Then, $\mathcal{S}$ performs the local check on all opened values and AEAD ciphertexts sent in the handshake phase following the protocol specification. $\mathcal{S}$ parses $\widetilde{\textsc{enc}_Q} = (\widetilde{\mathbf{C}_Q}, \widetilde{\sigma_Q})$ and checks that $\mathbf{C}_Q = \widetilde{\mathbf{C}_Q}$. Furthermore, $\mathcal{S}$ checks correctness of $\widetilde{\sigma_Q}$ following the protocol specification. Then, $\mathcal{S}$ parses $\textsc{enc}_R{}' = (\mathbf{C}_R', \sigma_R')$ and checks that $\mathbf{C}_R' = \mathbf{C}_R$. $\mathcal{S}$ also checks $(\textsc{h}_R, \sigma_R) = (\textsc{h}_R', \sigma_R')$. If any check fails, $\mathcal{S}$ aborts.
15. $\mathcal{S}$ emulates the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on $\mathsf{pms}_{\mathcal{P}}'$ stored previously and circuit $\mathit{AddModp}$ to compute $\mathsf{pms}^* = \mathit{AddModp}(\mathsf{pms}_{\mathcal{P}}', \mathsf{pms}_{\mathcal{V}})$.
16. Given pms^*, simulator $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$ for the post-record phase of sub-protocol executions $\Pi_{\mathsf{PRF}}^{(1)}$, $\Pi_{\mathsf{PRF}}^{(2)}$, $\Pi_{\mathsf{PRF}}^{(3)}$, $\Pi_{\mathsf{PRF}}^{(4)}$, $\Pi_{\mathsf{AEAD}}^{(1)}$, $\Pi_{\mathsf{AEAD}}^{(2)}$ and $\Pi_{\mathsf{AEAD}}^{(3)}$ to check the correctness of all values obtained by $\mathcal{V}$. If the check fails, then $\mathcal{S}$ aborts.

17. $\mathcal{S}$ emulates the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by computing $\textsc{z}_R := \mathsf{AES}(\mathsf{key}^*_S, \mathsf{st}^C_d)$ and receiving local keys on $[\![\textsc{z}_R]\!]$ from $\mathcal{A}$. After receiving $\textsc{z}'_R$ from $\mathcal{A}$, simulator $\mathcal{S}$ emulates the (check) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on $[\![\textsc{z}_R]\!] - \textsc{z}'_R$ by sending true to $\mathcal{A}$ if $\textsc{z}'_R = \textsc{z}_R$ or false to $\mathcal{A}$ otherwise. Following the protocol specification, $\mathcal{S}$ uses $(h_C, \textsc{z}_C, \textsc{z}_Q)$ to check correctness of all GMAC tags in AEAD ciphertexts $\textsc{fin}_C, \textsc{enc}_Q$, and aborts if the check fails. If $\textsc{fin}_S{}' \neq \textsc{fin}_S$ or $\textsc{enc}_R{}' \neq \textsc{enc}_R$, then $\mathcal{S}$ aborts.

18. $\mathcal{S}$ emulates the ($\mathsf{authinput}$) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving local keys on $[\![\textsc{z}_{i,\mathcal{P}}]\!]$ for $i \in [1, n]$ from $\mathcal{A}$. Then, $\mathcal{S}$ computes local keys on $[\![\textsc{z}_i]\!] = [\![\textsc{z}_{i,\mathcal{P}}]\!] \oplus \textsc{z}_{i,\mathcal{V}}$ for $i \in [1, n]$. Next, $\mathcal{S}$ computes local keys on $[\![\textsc{m}_i]\!] := [\![\textsc{z}_i]\!] \oplus \textsc{c}_i$ for $i \in [1, n]$ where $\mathbf{C}_Q = (\textsc{c}_1, \ldots, \textsc{c}_n)$ is the AES ciphertext included in $\textsc{enc}_Q$ and $[\![Q]\!] = ([\![\textsc{m}_1]\!], \ldots, [\![\textsc{m}_n]\!])$.

19. $\mathcal{S}$ emulates the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving local keys on $[\![R]\!]$.

20. $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{Conv}}$ to convert $([\![Q]\!], [\![R]\!])$ into additively homomorphic commitments by sending commitment identifiers to $\mathcal{A}$.

□