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 additively homomorphic 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 a prover $\mathcal{P}$ (i.e., a user that intends to prove the originality of the data) and a verifier $\mathcal{V}$ (i.e., a third party) to jointly emulate the computation of

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

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

the TLS client $\mathcal{C}$ who interacts with $\mathcal{S}$. Since neither $\mathcal{P}$ nor $\mathcal{V}$ ever holds TLS session keys, their capability is the same as man-in-the-middle attackers and thus cannot forge a valid TLS transcript for unauthorized data. In DECO, most of $\mathcal{C}$'s computation is emulated using a maliciously secure Two-Party Computation (2PC) protocol, which ensures that no derivation from the protocol can help the malicious party break the privacy or integrity requirement when interacting with $\mathcal{S}$. To prove statements on the TLS payload, $\mathcal{P}$ proves to $\mathcal{V}$ the correct decryption of the ciphertext (to obtain a plaintext) and desired statements on the plaintext using zkSNARKs [Gro10, GW11, BCCT12, BCTV14].

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, HSS17, 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, CWYY23], the computation and communication cost of maliciously secure 2PC is still much higher than its semi-honest counterparts. Moreover, these protocols with malicious security often require storing preprocessed authenticated triples, thus incurring a huge memory overhead. The complexity of the maliciously secure protocol also makes it difficult to implement and deploy such a protocol. As a result, the DECO protocol still requires 475 MB of communication to authenticate a 2KB-sized payload via TLS and more than 50 seconds to finish under a WAN network.

1.1 Our Contribution

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

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

function f. If $\mathcal{P}$ cheats, it can replace the function to one of its own choice but $\mathcal{V}$ cannot cheat in any way. During the checking phase, $\mathcal{P}$ will be given $x_{\mathcal{V}}$ and $\mathcal{V}$ should be notified if $\mathcal{P}$ cheated during the evaluation phase.

To accomplish this task, $\mathcal{P}$ first sends $\mathcal{V}$ a garbled circuit for f; they also use an OT with malicious security to let $\mathcal{V}$ get garbled labels on its input. Two parties then can obtain their outputs but there is no way to ensure correctness. For that, we ask $\mathcal{P}$ to commit to $\mathcal{V}$ its input $x_{\mathcal{P}}$ and output $y_{\mathcal{P}}$. Now, $\mathcal{V}$ has shares $x_{\mathcal{V}}, y_{\mathcal{V}}$ and commitments of $x_{\mathcal{P}}, y_{\mathcal{P}}$. After $\mathcal{P}$ gets $x_{\mathcal{V}}$, thus also $y_{\mathcal{V}}$, $\mathcal{P}$ can use a Zero-Knowledge (ZK) protocol to prove that $(y_{\mathcal{P}}, y_{\mathcal{V}}) = f(x_{\mathcal{P}}, x_{\mathcal{V}})$ w.r.t. the committed values. $\mathcal{P}$ could launch a selective failure attack on $x_{\mathcal{V}}$ (leaking one-bit of information), but it is meaningless since $x_{\mathcal{V}}$ is always given to $\mathcal{P}$ in the proving phase. For obvious reasons, we refer to this technique as *garble-then-prove*. This technique can be also applied in, e.g., QUIC [CDH+16, IT21], OAuth [Har12] and OpenID Connect [SBJ+14], to authenticate web data.

TLS-specific protocol optimization. Building on the above idea, we further optimize other TLS building blocks in various ways. For example, we show how to carefully select values to reveal, without providing any party an extra capacity, during the derivation of TLS session keys, leading to a more than 2-fold reduction in handshake circuit size. We also pull the computation of the Galois Message Authentication Code (GMAC) tags out of circuits and instead use Oblivious Linear Evaluation with errors (OLEe) [1] to compute additive sharings of the powers of a random element needed for GMAC, reducing the cost of GMAC computation by more than two orders of magnitude. Doing so would allow the adversary to gain one bit of information of the TLS session key, but that would not reduce the overall concrete security, for a reason similar to prior works, e.g., [KOS15, YWZ20].

Efficient commitment conversion. To prove statements on the TLS data using zkSNARKs, DECO embeds the TLS ciphertext into the statements and then proves in ZK the correctness of decryption. For our protocol, we use the recent vector OLE (VOLE) based interactive zero-knowledge proofs [WYKW21, DIO21, BMRS21, YSWW21] during the garble-then-prove execution. This means that at the end of the protocol, two parties hold information-theoretic MACs (IT-MACs) on each bit of the query and response involved in the TLS. One could prove statements using VOLE-based ZK proofs or, alternatively, convert them to commitments friendly to zkSNARKs. First, we convert IT-MACs over $\mathbb{F}_2$ to IT-MACs over $\mathbb{F}_q$, ensuring the values are consistent. This protocol can be viewed as a special version of zero-knowledge via garbled-circuit protocol [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 zkSNARK-friendly commitments, which can be achieved with high efficiency, since both representations are additive-homomorphic. In this way, without using zkSNARKs, we can convert the plaintext query and response to additively homomorphic commitments, which can then be connected to various zkSNARKs, e.g., [CFQ19, CHM+20, 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

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

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.

2 Preliminaries

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

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

We use $[x]_p = (x_\mathcal{P}, x_\mathcal{V})$ to denote an additive secret sharing of x over $\mathbb{Z}_p$ between $\mathcal{P}$ and $\mathcal{V}$ holding $x_\mathcal{P}$ and $x_\mathcal{V}$ respectively. When the field is $\mathbb{F}_{2^{128}}$, we denote by $[x]_{2^{128}}$. For details of additive secret sharings, we refer to the reader for Section A.2. Let $[\![x]\!] = (x, \mathsf{M}[x], \mathsf{K}[x])$ be an Information-Theoretic Message Authentication Code (IT-MAC) such that $\mathsf{M}[x] = \mathsf{K}[x] + x \cdot \Delta$, where the message x and MAC tag $\mathsf{M}[x]$ are held by a party $\mathcal{P}$, and keys $\mathsf{K}[x], \Delta$ are obtained by another party $\mathcal{V}$. We give more details of IT-MACs in Section A.3.

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 Section 4.3, where both of TLS 1.2 and TLS 1.3 adopt HMAC to derive secrets and keys. [2] While TLS provides different modes, we focus on the following most popular modes:

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

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

Our protocol is easy to be extended to support that AEAD scheme is instantiated by $\mathsf{AES256\_GCM}$ and H is replaced with $\mathsf{SHA384}$, and also allows one to use other digital signature (e.g., DSA). Besides, our protocol can be straightforwardly extended to support ECDH in which the server uses a static DH value (rather than an ephemeral DH value). We did not optimize our protocol to realize the CBC mode in TLS 1.2, since this mode has been demonstrated to be vulnerable to the timing attack against several TLS implementations [AP13], and the GCM mode is preferred

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

over CBC [tlsb]. In addition, TLS 1.3 did not support the CBC mode any more. Our garble-then-prove approach can be also generalized to other modes such as CHACHA20_POLY1305_SHA256 and AES128_CCM_SHA256. In Section 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 pseudo-random function HMAC is defined as follows:

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

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

Key derivation. Here we focus on the Pseudo-Random Function (PRF) in TLS 1.2 [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 Section 4.3 for the details of HKDF. Specifically, the PRF function with output length ℓ in TLS 1.2 is defined below:

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

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

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

1. Compute $\mathrm{Z}_0 := \mathsf{AES}(\mathsf{key}, \mathsf{st}_e)$ and update $\mathsf{st}_e := \mathsf{st}_e + 1$.
2. Suppose $\mathbf{M}$ is padded as $(\mathrm{M}_1, \ldots, \mathrm{M}_n)$ with $\mathrm{M}_i \in \{0,1\}^{128}$. From $i = 1$ to n, compute $\mathrm{Z}_i := \mathsf{AES}(\mathsf{key}, \mathsf{st}_e)$ and $\mathrm{C}_i := \mathrm{Z}_i \oplus \mathrm{M}_i$ and update $\mathsf{st}_e := \mathsf{st}_e + 1$. Set $\mathbf{C} := (\mathrm{C}_1, \ldots, \mathrm{C}_n)$.
3. Suppose that the header H has been padded as an element in $\mathbb{F}_{2^{128}}$. Let ℓ_{H} be the bit length of H. Given a vector $\mathbf{X} \in (\mathbb{F}_{2^{128}})^m$, the GHASH polynomial $\Phi_{\mathbf{X}} : \mathbb{F}_{2^{128}} \to \mathbb{F}_{2^{128}}$ is defined as $\Phi_{\mathbf{X}}(h) = \sum_{i=1}^{m} X_i \cdot h^{m-i+1} \in \mathbb{F}_{2^{128}}$. Compute $h := \mathsf{AES}(\mathsf{key}, \mathbf{0})$ and a GMAC tag $\sigma := \mathrm{Z}_0 \oplus \Phi_{(\mathrm{H}, \mathbf{C}, \ell_{\mathrm{H}}, \ell_{\mathrm{C}})}(h)$.
4. Output $\mathrm{CT} = (\mathbf{C}, \sigma)$.

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

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

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.**

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), OLEe and commitments in Section A.1. We also describe the definition of ideal functionality for interactive ZK proofs based on IT-MACs in Section A.4.

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 TLS server $\mathcal{S}$, where $\mathcal{P}$ and $\mathcal{V}$ jointly play the role of the client to interact with the server $\mathcal{S}$. Prover $\mathcal{P}$ has data stored on the server, and intends to prove to $\mathcal{V}$ about properties of the data, without any modification to the TLS server. At a high level, the protocols to authenticate web data will involve the following steps performed in a secure and distributed way:

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

In this paper, we focus on constructing a secure protocol to realize the first three steps. The final step can be realized using a variety of zero-knowledge proofs such as zk-SNARKs [Gro10, GW11, BCCT12]. 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].

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

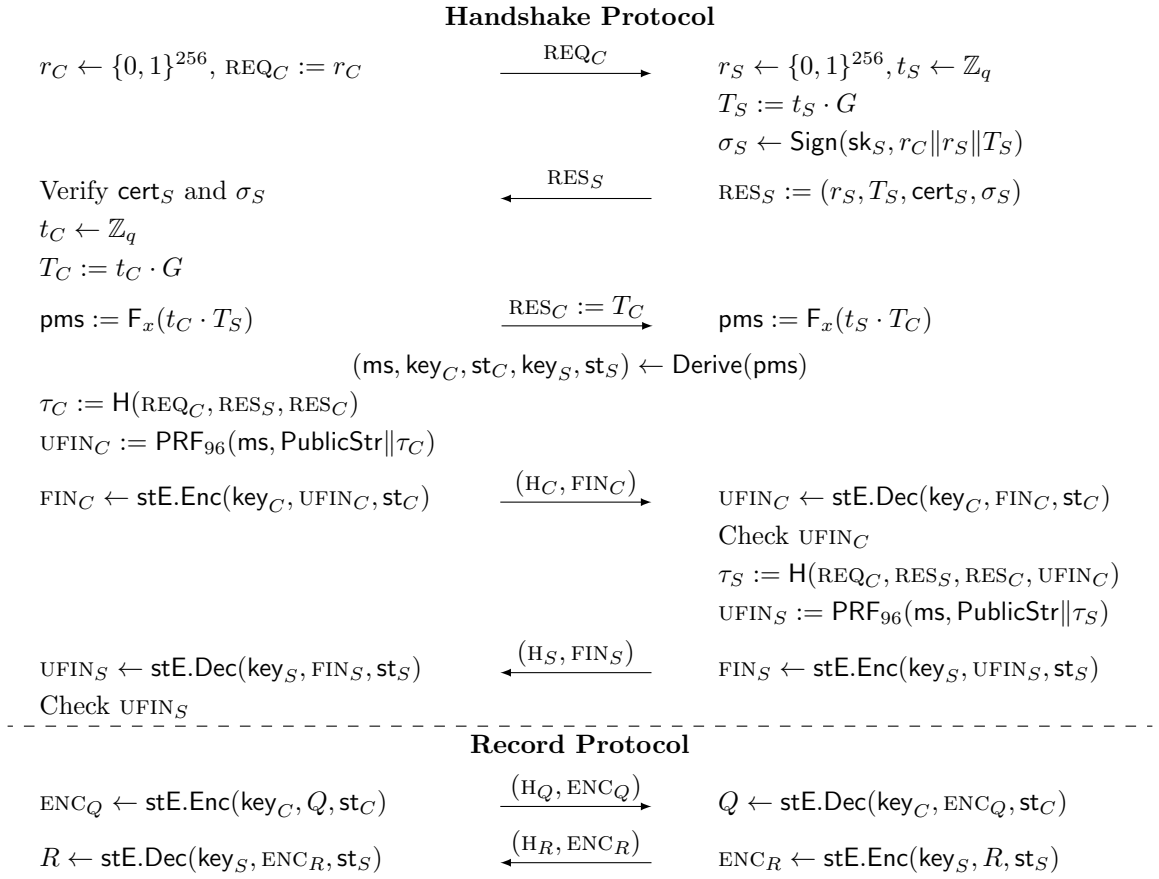

Figure 2: **Graphical depiction of TLS.** PublicStr refers to strings defined in the TLS specification. H_C, H_S, H_Q and H_R are public metadata headers defined by the TLS specification. Some details are omitted.

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

3 Technical Overview

Third-party authentication of TLS payload could be achieved using a malicious 2PC protocol with a high overhead [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 Two-Party Computation protocols based on Garbled Circuits (GC-2PC) that are the same as semi-honest protocols [Yao86, ZRE15, RR21] except for using malicious OT instead of semi-honest OT, and the recent VOLE-based 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 could also be used in 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.

3.1 An Overview of the TLS Protocol

In Figure 2, we provide a pictorial overview, and show complete details in Figures 9 and 10 of Section 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.** $\mathcal{C}$ samples a random nonce $r_C \leftarrow \{0,1\}^{256}$, and then sends $\text{REQ}_C = r_C$ to $\mathcal{S}$. Then, $\mathcal{S}$ samples a random nonce $r_S \in \{0,1\}^{256}$, a random element $t_S \in \mathbb{Z}_q$, and computes a group element $T_S := t_S \cdot G$. $\mathcal{S}$ sends back $\text{RES}_S = (r_S, t_S, \mathsf{cert}_S, \sigma_S)$, where cert_S is a certification and σ_S is a signature on (r_C, r_S, T_S). To finish the key-exchange protocol, $\mathcal{C}$ sends back a random group element $T_C := t_C \cdot G$. Now both parties agree on a pre-master secret $\mathsf{pms} = \mathsf{F}_x(t_C \cdot T_S) = \mathsf{F}_x(t_S \cdot T_C)$, where F_x is a function mapping an elliptic-curve point to its x-coordinate.
- **Phase 2: TLS session keys.** With pms, $\mathcal{C}$ and $\mathcal{S}$ compute a master secret $\mathsf{ms} := \mathsf{PRF}_{384}(\mathsf{pms}, \text{“master secret”}, r_C \| r_S)$. Then, both parties compute a tuple $(\mathsf{key}_C, IV_C, \mathsf{key}_S, IV_S) := \mathsf{PRF}_{448}(\mathsf{ms}, \text{“key expansion”}, r_S \| r_C)$, where $\mathsf{key}_C, \mathsf{key}_S \in \{0,1\}^{128}$ are two application keys and $IV_C, IV_S \in \{0,1\}^{96}$ are the initial states $\mathsf{st}_C, \mathsf{st}_S$ of AEAD encryption. In Figure 2, we refer to the whole process as Derive.
- **Phase 3: Finished messages.** Two parties exchange test messages, which have already been known by them, over the established AEAD-encrypted channel. The client's message is $\text{UFIN}_C = \mathsf{PRF}_{96}(\mathsf{ms}, \text{“client finished”}, \tau_C)$, where τ_C is the hash of the TLS transcripts so far. $\mathcal{C}$ sends the AEAD ciphertext FIN_C of this message, which is encrypted with key_C and st_C, to $\mathcal{S}$. The server decrypts FIN_C and checks if UFIN_C is correct based on the same session key and values. Then $\mathcal{S}$ sends back a similarly encrypted message, and $\mathcal{C}$ checks its correctness.
- **Phase 4: Exchange payload.** Finally, two parties exchange their application payload. The exact process is essentially the same as Phase 3, with updated states for AEAD (based on AES-GCM), except that now the underlying payload is provided by the client and server based on the application. This phase could exchange several rounds of payload, depending on the application. In Figure 2, we only show one round of payload for simplicity. The following technical overview

focuses on the one-round case, and our protocol can be extended to support multiple rounds of payload (see Section 4.2 for details).

3.2 Our Protocol Design

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

3.2.1 Phase 1: Generating pre-master secret

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

Obtaining fully secure OLE is often expensive and requires tailored zero-knowledge proofs or excessive communication. However, in this particular setting, we show that an OLE with error (OLEe), where the error could even depend on parties' inputs, is already sufficient. Such an OLEe can be efficiently computed using $\log q$ correlated OTs without the need of any extra checks. This would lead to one-bit information leakage about pms to the adversary who corrupts the prover $\mathcal{P}$. However, due to the TLS protocol, pms is of high entropy and we can show that such leakage does not help the adversary in guessing the whole secret pms. Intuitively, such an error could only lead to the selective-failure attack, which allows the adversary to guess c bits of the secret with probability 2^{-c}, but if the guess is incorrect the protocol execution aborts. Such an attack does not reduce concrete security since the adversary could bet on c bits of the secret too. A similar analysis has already been used in designing maliciously secure protocols (e.g., [KOS15, CDE+18, YWZ20]).

3.2.2 Phase 2: Deriving TLS session keys

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

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

a seemingly catastrophic way: a malicious $\mathcal{P}$ could change a Garbled Circuit (GC) to control the output to be anything (could even be pms or something that can help $\mathcal{P}$ recover pms).

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

This optimization alone significantly reduces the overhead of the protocol as it eliminates the need of a malicious 2PC protocol, which is expensive in computation/communication but also requires memory linear to the circuit size to store the preprocessing triples. We formally model the 2PC with garble-then-prove approach as an ideal functionality $\mathcal{F}_{\mathsf{GP2PC}}$ shown in Section B.1, and show how to instantiate $\mathcal{F}_{\mathsf{GP2PC}}$ using plain GC-2PC with malicious OT and interactive ZK, which is described in Section B.2. The 2PC protocol with garble-then-prove approach may be of independent interest, and may be applied in other scenarios such that all $\mathcal{V}$'s secrets are allowed to be revealed to $\mathcal{P}$ at the end.

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

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

In the above equation, $\mathsf{HMAC}(k, m) = \mathsf{SHA256}(k \oplus \text{opad}, \mathsf{SHA256}(k \oplus \text{ipad}, m))$, and that $\mathsf{SHA256}(m_1, m_2, m_3) = f_{\mathsf{H}}(f_{\mathsf{H}}(f_{\mathsf{H}}(IV_0, m_1), m_2), m_3)$ where m_i's are 512-bit strings. To compute an HMAC-SHA256, we need at least 4 SHA256 compress calls: 2 calls to compute the outer hash and at least 2 calls to compute the inner hash; if m is longer than 447 bits, the inner hash requires even more calls.

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

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

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

3.2.3 Phase 3: Finished messages

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

AES-GCM computation consists of two tasks: computing the ciphertext and computing the GMAC tag. The first task is relatively easy as we can use the garble-then-prove approach again to avoid malicious 2PC, where the plaintext is known by both parties in this phase. However, computing the GMAC tag is more complicated. Roughly speaking, the GMAC tag is an inner product between a public vector over $\mathbb{F}_{2^{128}}$ and a private vector $(\text{Z}_0, h^1, \ldots, h^n)$ where $\text{Z}_0 = \mathsf{AES}(\mathsf{key}, \mathsf{st})$ is shared by both parties. Revealing any term in the second vector would allow the adversary to forge a GMAC tag on any message of its choice. Computing Z_0 can be done in GC-2PC; however, since we reveal the additive shares of Z_0, meaning that the output is not well defined from $\mathcal{V}$'s perspective, the garble-then-prove approach does not immediately work. To solve this issue, we ask $\mathcal{P}$ to commit to its share of Z_0. After the completion of the TLS protocol, when $\mathcal{P}$ knows all secrets, $\mathcal{P}$ will prove the computation with respect to the above commitment. To avoid computing h^i in circuits, we also reveal the additive shares of h together with Z_0. Then two parties use an OLEe over $\mathbb{F}_{2^{128}}$ to compute additive sharings on all powers of h. This way, each term only needs *2KB* communication, 100× smaller than computing in GC-2PC! Similar to the use of OLEe in phase 1, this also introduces a chance of a selective failure attack; however, it can be easily shown that providing multiple chances of selective failure attacks does not provide any more power to the adversary.

3.2.4 Phase 4: Payload

This phase is the first time $\mathcal{P}$ provides a private input (namely the query string) that is not part of the TLS execution. The overall protocol is similar to phase 3 how we compute the finished messages, except that the plaintext to AES-GCM-based AEAD is not public anymore. Therefore,

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

we can mostly follow the phase-3 protocol except that $\mathcal{P}$ XOR its query to the additive share of AES output, and then sends the resulting value to $\mathcal{V}$. In this way, $\mathcal{V}$ can obtain the ciphertext directly by XORing the resulting value with its additive share.

After obtaining the AEAD cihertexts ENC_Q and ENC_R on the query Q and response R from $\mathcal{P}$, $\mathcal{V}$ opens $t_{\mathcal{V}} \in \mathbb{Z}_q$ to $\mathcal{P}$, who can replay the whole TLS protocol to obtain all values computed in GC-2PC. At this point, $\mathcal{V}$ holds 1) the commitment to $\mathcal{P}$'s share of pms; 2) the commitments to all values revealed from GC-2PC as XOR shares of AES outputs; 3) the values revealed from GC-2PC to both parties. Now $\mathcal{P}$ can prove to $\mathcal{V}$ in zero-knowledge that the whole computation is correct with respect to the commitments and values that $\mathcal{V}$ has. The circuit proven in ZK includes 1) the circuit computed in GC-2PC and 2) the decryption of the ciphertext to the response. However, the cost of ZK is significantly smaller than GC-2PC: when using the latest VOLE-based ZK [YSWW21], the communication of ZK is only 1 bit per AND gate, compared to 256 bits per AND gate required by the GC-2PC protocol [ZRE15]. During the process of ZK, $\mathcal{P}$ also needs to commit to the plaintext of the query and response to prove AEAD computation. They will be converted to a ZK-friendly format in the next phase.

3.2.5 Converting to ZK-Friendly Commitments

Now $\mathcal{V}$ has commitments of the query Q and response R that $\mathcal{P}$ knows. Their correctness has been verified by $\mathcal{V}$ through VOLE-based ZK. Such commitments are instantiated by IT-MACs and denoted by $[\![\boldsymbol{u}]\!] = ([\![u_1]\!], \ldots, [\![u_\ell]\!])$, where for each $i \in [1, \ell]$, $u_i \in \{0,1\}$, $(u_i, \mathsf{M}[u_i])$ is obtained by $\mathcal{P}$ and $(\mathsf{K}[u_i], \Delta)$ is 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, $\mathcal{V}$ computes $\widetilde{\mathsf{K}}[u_i] := \mathsf{H}(\mathsf{K}[u_i])$ and sends $W_i := \mathsf{H}(\mathsf{K}[u_i]) - \mathsf{H}(\mathsf{K}[u_i] \oplus \Delta) + \Gamma \in \mathbb{Z}_q$ to $\mathcal{P}$, who computes $\widetilde{\mathsf{M}}[u_i] := \mathsf{H}(\mathsf{M}[u_i]) + u_i \cdot W_i = \widetilde{\mathsf{K}}[u_i] + u_i \cdot \Gamma$, where $\Gamma \in \mathbb{Z}_q$ is a uniform global key known to $\mathcal{V}$. We also ask $\mathcal{P}$ to commit to (Q, R) using an additively homomorphic commitment (e.g., Pedersen [Ped92] and KZG [KZG10]) that is friendly to zkSNARKs. To check consistency between IT-MACs over $\mathbb{Z}_q$ and additively homomorphic commitments, we reveal a random linear combination of the values committed in two formats, where the random challenges are chosen by $\mathcal{V}$.

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

3.3 Protocol Summary

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

1. $\mathcal{P}$ samples and sends REQ_C to $\mathcal{S}$ and gets back RES_S.

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

4 Authenticating Web Data for TLS

In Section B, we define an ideal functionality $\mathcal{F}_{\mathsf{GP2PC}}$ for 2PC in the garble-then-prove framework, and then show an efficient protocol securely realizing $\mathcal{F}_{\mathsf{GP2PC}}$. Based on $\mathcal{F}_{\mathsf{GP2PC}}$ we provide a complete description of our protocol (denoted by Π_{AuthData}) that authenticates web data for TLS 1.2 in Section 4.1. Then, we show how to extend protocol Π_{AuthData} to support multi-round query-response sessions and describe further optimizations in Section 4.2. While Π_{AuthData} focuses on the case of reading user data, we also extend it to support for writing user data in Section 4.2. In Section 4.3, we also show how to extend our protocol to support TLS 1.3.

4.1 Detailed Protocol for Authenticating Data

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

- **Preprocessing:** A prover $\mathcal{P}$ and a verifier $\mathcal{V}$ generate correlated randomness before the TLS connection.

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

The main protocol Π_{AuthData} invokes the following three sub-protocols, whose details are described in Section C.

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

$\mathcal{P}$ and $\mathcal{V}$ generate authenticated bits $[\![Q]\!]$ and $[\![R]\!]$ in the post-record phase by calling an ideal functionality $\mathcal{F}_{\mathsf{ZK}}$ for ZK proofs based on IT-MACs. Functionality $\mathcal{F}_{\mathsf{ZK}}$ (shown in Section A.4) is a simple extension of the ideal functionality defined in [WYX+21], and can be securely realized using the recent VOLE-based ZK protocols [WYKW21, DIO21, BMRS21, YSWW21]. Besides, $\mathcal{P}$ and $\mathcal{V}$ call an ideal functionality $\mathcal{F}_{\mathsf{Conv}}$ (shown in Section D) to convert $[\![Q]\!]$ and $[\![R]\!]$ into their additively homomorphic commitments in the post-record phase. In Section D, we present an efficient protocol to securely realize $\mathcal{F}_{\mathsf{Conv}}$.

We postpone the details of protocol Π_{AuthData} to Figures 17 and 18 in Section C.4. 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]. For one-round session, $\mathcal{P}$ is *unnecessary* to decrypt the AEAD ciphertext ENC_R on the response R and verify its GMAC tag by running sub-protocol Π_{AEAD} with $\mathcal{V}$. These operations can be performed *locally* by $\mathcal{P}$ after it knows the server-to-client key key_S, where the TLS session terminates after ENC_R was received by $\mathcal{P}$ and forwarded to $\mathcal{V}$. Nevertheless, $\mathcal{P}$ and $\mathcal{V}$ still need to generate $[\![\text{z}_R]\!]$ and $[\![R]\!]$ by calling functionality $\mathcal{F}_{\mathsf{ZK}}$. $\mathcal{V}$ also needs to check the correctness of the GMAC tag in ciphertext ENC_R via getting $h_S = \mathsf{AES}(\mathsf{key}_S, \mathbf{0})$ and $\text{z}_R = \mathsf{AES}(\mathsf{key}_S, \mathsf{st}_d^C)$.

The security of protocol Π_{AuthData} depends on the PRF-Oracle-Diffie-Hellman (PRF-ODH) assumption, which has been used for proving the security of TLS 1.2 [JKSS12, KPW13] and is recalled in Section E. Besides, we assume that the underlying signature scheme satisfies Existential UnForgeability under Chosen-Message Attack (EUF-CMA).

Theorem 1. *If the PRF-ODH assumption holds and the underlying signature scheme is EUF-CMA secure, then protocol* Π_{AuthData} *(shown in Figures 17 and 18) 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 Theorem 1 in Section E.

4.2 Extensions and Optimizations

Extend to multi-round query-response sessions. We are able to extend the protocol Π_{AuthData} to support multiple rounds of payload. Specifically, $\mathcal{P}$ and $\mathcal{V}$ can execute sub-protocol Π_{AEAD} (shown in Figures 15 and 16 of Section C.3) multiple times to encrypt multiple queries, where the additive sharings of powers of $h_C = \mathsf{AES}(\mathsf{key}_C, \mathbf{0})$ need to be computed only once and are reused among these sub-protocol executions. Note that the state st_C is always increased for computing multiple AEAD ciphertexts following the TLS specification. This prevents $\mathcal{P}$ or $\mathcal{V}$ to forge GMAC tags by using the same state for different ciphertexts.

If every query is independent of previous responses (Case 1), $\mathcal{P}$ can *locally* decrypt the AEAD ciphertexts of all responses, after the TLS session terminates and it obtains the server-to-client application key key_S. If every query relies on previous responses (Case 2), $\mathcal{P}$ has to decrypt the ciphertexts of all responses via interacting with $\mathcal{V}$. This can be done by running sub-protocol Π_{AEAD} with type_1 = "decryption" and type_2 = "secret", where Π_{AEAD} was designed for supporting decryption of AEAD ciphertexts in the record phase. During the protocol execution, Π_{AEAD} also allows $\mathcal{P}$ and $\mathcal{V}$ to verify the correctness of GMAC tags in AEAD ciphertexts of responses. Therefore, in both cases, $\mathcal{P}$ can check the correctness of AEAD ciphertext on every response via sending the ciphertext to $\mathcal{V}$ and then running sub-protocol Π_{AEAD} with $\mathcal{V}$, before generating the ciphertext on the next query. In fact, this is unnecessary and the GMAC tags of AEAD ciphertexts on all responses can be verified *locally* by $\mathcal{P}$ after it knows key_S (see below for discussion of this optimization). In Case 2, the decryption of the response's ciphertext sent in the final-round session can still be performed *locally* with key_S.

In the case of multi-round sessions, both $\mathcal{P}$ and $\mathcal{V}$ can use the same approach implied in the post-record phase of main protocol Π_{AuthData} (Figures 17 and 18) to check the correctness of all AEAD ciphertexts on multiple queries and responses. In Case 2, we note that $\mathcal{P}$ needs to decrypt the AEAD ciphertexts on responses using key_S, and then compares the resulting plaintexts with that obtained during the execution of sub-protocol Π_{AEAD}, when it performs the local verification with key_S in the post-record phase. This verification aims to check that no error is introduced to the responses computed via the distributed decryption in sub-protocol Π_{AEAD}. Both parties are also able to obtain the IT-MACs on all responses in a way totally similar to main protocol Π_{AuthData}. Note that the IT-MACs on all queries have already been obtained during multiple executions of sub-protocol Π_{AEAD}.

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

1. $\mathcal{P}$ sends Z_1 to $\mathcal{V}$, and then proves $\text{Z}_1 = \mathsf{AES}(\mathsf{key}_S^*, IV_S + 1)$ by calling the (prove) command of$\mathcal{F}_{\mathsf{GP2PC}}$, where key_S^* and IV_S are the server-to-client application key and initial vector whose correctness has been proved in the post-record phase. Then, $\mathcal{V}$ checks that $\text{UFIN}_S \oplus \text{Z}_1 = \text{C}$.
2. $\mathcal{V}$ checks the correctness of GMAC tag σ in the way shown in the post-record phase of protocol Π_{AuthData}.

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

Extend to support for writing user data. Our protocol Π_{AuthData} (shown in Figures 17 and 18) focuses on reading user data from a website acting as the TLS server. For most of Web2 and Web3 applications, it is sufficient. Nevertheless, for a few applications, a user (i.e., prover) may be desirable to write its data on the website (e.g., updating personal information) during the protocol execution of Π_{AuthData}. In this case, a malicious verifier $\mathcal{V}$ may tamper the queries sent from a prover $\mathcal{P}$ to the TLS server by adding some errors into the AES ciphertexts on queries. The attack would be detected by $\mathcal{P}$ after it obtains all secrets. However, the user data on the website has already been tampered. To prevent such attacks, we need to extend functionality $\mathcal{F}_{\mathsf{GP2PC}}$ to an ideal functionality $\mathcal{F}^{\mathsf{noerr}}_{\mathsf{GP2PC}}$ (defined in Section B.1) that does not allow $\mathcal{V}$ to introduce any errors. In Section B.2, we show how to extend the protocol instantiating $\mathcal{F}_{\mathsf{GP2PC}}$ to securely realize $\mathcal{F}^{\mathsf{noerr}}_{\mathsf{GP2PC}}$ with no extra communication. When replacing $\mathcal{F}_{\mathsf{GP2PC}}$ with $\mathcal{F}^{\mathsf{noerr}}_{\mathsf{GP2PC}}$, protocol Π_{AuthData} would allow $\mathcal{P}$ to securely write user data.

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

4.3 Extending Our Protocol for TLS 1.3

While the protocol Π_{AuthData} shown in Figures 17 and 18 of Section C.4 focuses on the case of TLS 1.2, we are also able to extend it for TLS 1.3, and will implement the protocol to authenticate web data for TLS 1.3 in the future work. The main differences between TLS 1.2 and TLS 1.3 are the key derivation function (KDF) and the handshake phase. In this section, we focus on the handshake mode of full 1-RTT, where the optional mode of 0-RTT based on a pre-shared key can be securely computed in a similar way.

The key derivation in TLS 1.3 adopts the HMAC-based key derivation function (HKDF) [Kra10, KE10], which consists of two sub-functions: $\mathsf{HKDF.Extract}$ and $\mathsf{HKDF.Expand}$. Specifically, $\mathsf{prk} \leftarrow \mathsf{HKDF.Extract}(\mathit{salt}, \mathsf{ikm})$ takes as input a non-secret random *salt* and a secret input key material ikm, and then extracts a pseudorandom key prk, i.e., $\mathsf{prk} = \mathsf{HMAC}(\mathit{salt}, \mathsf{ikm})$. Note that *salt* is the HMAC key, and ikm is the HMAC input. In this case, we can securely compute $\mathsf{HKDF.Extract}$ in the following manner:

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

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

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

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

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

While chk and shk are used to encrypt/decrypt the subsequent messages (e.g., client/server finished messages, signatures and certifications) in the handshake phase, cak and sak are independent and used to encrypt/decrypt application data in the record phase. Therefore, we can open chk and shk to the prover $\mathcal{P}$, and then $\mathcal{P}$ is able to *locally* perform the encryption/decryption in the handshake phase. [5] That is, it is unnecessary to run sub-protocol Π_{AEAD} (shown in Section C.3) to compute stateful AEAD in the handshake phase. Besides, handshake traffic secrets chts and shts are used to produce client and server finished messages, and are independent from application traffic secrets cats and sats. In this case, we can open chts and shts to $\mathcal{P}$, and then $\mathcal{P}$ can *locally* generate the finished messages. Furthermore, ems can be computed *locally* by $\mathcal{P}$ after it knows all secrets in the post-record phase, as ems is an exporter master secret and *not* used in the record phase. The secret rms is also able to be computed *locally* by $\mathcal{P}$ after it knows all secrets in the post-record phase, if $\mathcal{P}$ and $\mathcal{V}$ will not jointly execute a session resumption with rms. The optimizations would significantly reduce the cost in the handshake phase. Overall, 21 invocations of SHA256 compression functions need to be executed in GC-2PC, compared to that 14 invocations in TLS 1.2 for our protocol and 30 invocations in TLS 1.3 for DECO [ZMM+20]. As for our protocols, the communication cost in the handshake phase of TLS 1.3 is about 1.5× larger than that in TLS 1.2.

5 Performance Evaluation

5.1 Implementation and Experimental Setup

We implemented our protocol in C++, including 4000 lines of code of protocol development and 3000 lines of testing code. Our implementation is complete and can interact with real-world APIs. We use the EMP toolkit [WMK16] for the implementation of the following building blocks: KOS OT [KOS15], Ferret OT [YWL+20], half-gates-based GC with optimization of concrete security [ZRE15, GKWY20] and interactive ZK (called QuickSilver) [YSWW21]. We leave it as the future work to incorporate the recent three-halves GC construction [RR21] to further reduce the communication cost of our protocol.

[5] This observation has been found in DECO [ZMM+20].

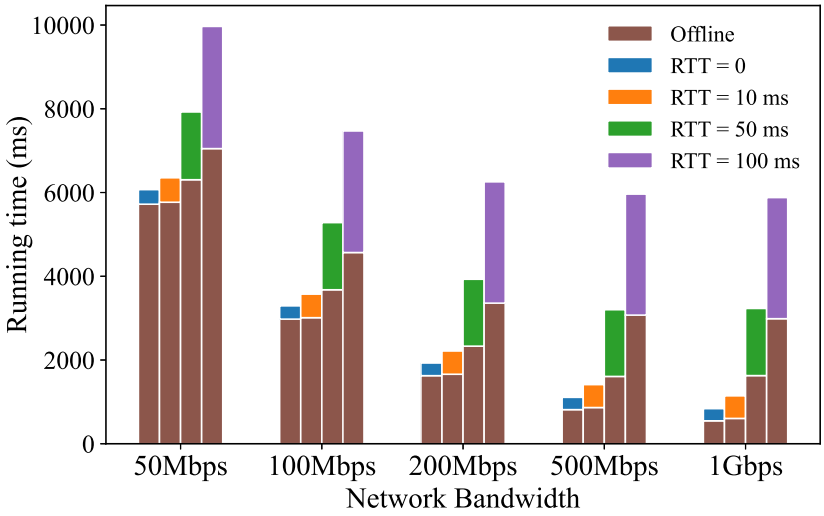

Figure 3: **Performance of our protocol under different network bandwidths and latency.** The length of query and response is 2 KB.

Payload	256B	512B	1KB	2KB	256B	512B	1KB	2KB	256B	512B	1KB	2KB
	Communication cost MegaByte (MB)				WAN (100Mbps, RTT=50ms) Second (s)				LAN (1Gbps, RTT=0ms) Second (s)			
DECO [ZMM+20]	206	255	345	475.7	24	27.2	36.3	51.6	5.91	6.46	8.9	11.21
This work	15.2	17.8	22.9	33.3	3.19	3.43	3.96	4.9	0.46	0.51	0.61	0.72

Table 2: **Comparing the performance of DECO [ZMM+20] and our protocol under LAN and WAN.**

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

5.2 Scalability of Our Protocol

Performance of protocol Π_{AuthData}. In Figure 3, we show the performance of our protocol Π_{AuthData} (shown in Figures 17 and 18) under different bandwidths and latency, while fixing the query and response to 2KB. We show both the offline cost (which can be done before the TLS connection) and the online cost (which can only be done during the TLS connection). Overall, our protocol is highly efficient. For example, under a realistic network with 200 Mbps bandwidth and 50 ms latency, the end-to-end running time is under four seconds while the runtime in the online phase is less than two seconds.

We can also see that the online performance is highly dependent on the latency: it is less than 50 ms when the latency is low, but could be up to 3 seconds when the latency is as high as 100 ms. This matches the roundtrip complexity that we measured from our implementation, which needs

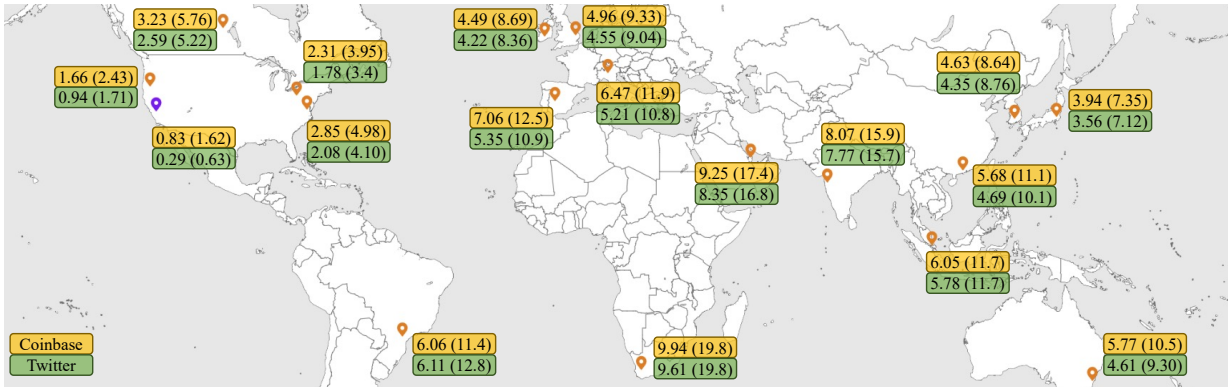

Figure 4: **Online and total performance of accessing Coinbase and Twitter servers with globally distributed provers.** All numbers are reported in seconds in the form of "online time (total time)". The verifier is fixed at California. The server is hosted by Coinbase/Twitter, which may have mirrors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

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

Table 3: **Performance of commitment conversion with different payload length.**

31 roundtrips of communication. The offline cost is less affected by the latency but more on the network bandwidth; this is because the transmission of garbled circuits, which is majority of the communication of our protocol, is in the offline phase.

Comparison with prior work. We compare the performance of our protocol with DECO [ZMM+20]. Since the code of DECO is not open sourced and that the performance of malicious 2PC has been constantly improving, we benchmark the performance based on the latest implementation of authenticated garbling. We also incorporate the Ferret OT [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 computing the TLS session keys and 4 AES-GCM ciphertexts. When computing the GMAC tag, we assume that one field multiplication over $\mathbb{F}_{2^{128}}$ takes 8,765 AND gates, including 8,192 ANDs to compute the multiplication and 573 ANDs to compute the reduction. Note that there exists more efficient garbling for binary extension field multiplication [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 instances. The result of the comparison is shown in Table 2, where we can observe roughly 14× improvement in communication and 7.5× to 15× improvement in running time over LAN and WAN.

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

Performance of conversion. We also benchmarked the performance of commitment conversion

of our protocol in different network settings, which is shown in Table 3. The IT-MAC-based commitments on payload is converted to Pedersen commitments [Ped92]. We observe that in both WAN and LAN settings, the conversion protocol is very cheap compared to the overall web authentication protocol, and the cost of conversion is linear to the payload size. It takes roughly 37 ms to convert an additional kilobyte of payload to Pedersen commitments under LAN and roughly 67 ms per KB under WAN. The basetime in WAN is higher due to the higher latency.

5.3 Global-Scale Benchmarks

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

- **Coinbase API:** We benchmark fetching the balance of BTC using the prover's API secret [coi]. It has a query of size 426 bytes and response of size 5701 bytes. Our protocol communicates 17.6 MB in the offline phase and 0.9 MB in the online phase.
- **Twitter API:** We benchmark using the prover's credential token to retrieve the number of followers [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 18 cities worldwide. All prover and verifier machines are hosted in AWS while the TLS server is hosted by Coinbase/Twitter, which may have nodes close to the prover. The online time required for the process ranges from 0.3 seconds to 10 seconds, depending on the round-trip time between the prover and verifier, which aligns with our expectation. From the experimental results shown in Figure 4, we conclude that our protocol is concretely efficient for real-world applications.

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

Acknowledgements

The authors would like to thank the members from TLSNotary for their helpful discussion. Kang Yang is supported by the National Natural Science Foundation of China (Grant Nos. 62102037 and 61932019). Yu Yu is supported by the National Natural Science Foundation of China (Grant Nos. 62125204 and 92270201), 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, #2310927 and research awards from Meta and Google. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[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.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In Shafi Goldwasser, editor, *ITCS 2012*, pages 326–349. ACM, January 2012.

[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.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 781–796. USENIX Association, August 2014.

[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.

[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.

[CWYY23] Hongrui Cui, Xiao Wang, Kang Yang, and Yu Yu. Actively secure half-gates with minimum overhead under duplex networks. In *EUROCRYPT 2023, Part II*, LNCS, pages 35–67. Springer, Heidelberg, June 2023.

[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.

[Gro10] Jens Groth. Short non-interactive zero-knowledge proofs. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 341–358. Springer, Heidelberg, December 2010.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In Lance Fortnow and Salil P. Vadhan, editors, *43rd ACM STOC*, pages 99–108. ACM Press, June 2011.

[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.

[HKE12] Yan Huang, Jonathan Katz, and David Evans. Quid-Pro-Quo-tocols: Strengthening semi-honest protocols with dual execution. In *2012 IEEE Symposium on Security and Privacy*, pages 272–284. IEEE Computer Society Press, May 2012.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

[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.

[JKSS12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 273–293. Springer, Heidelberg, August 2012.

[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.

[KPW13] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 429–448. Springer, Heidelberg, August 2013.

[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.

[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, 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

Ideal/real security model. 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

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

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

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

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

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

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

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

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 5. Correlated OT (COT) is an important variant of OT where two messages m_0 and m_1 satisfy a fixed correlation, i.e., $m_0 \oplus m_1 = \Delta$. Both OT and COT correlations can be generated in the malicious setting using either the IKNP-like protocols [KOS15, Roy22] or the PCG-like protocols [BCG+19b, BCG+19a, YWL+20].

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

Functionality for OLE with errors is shown in Figure 6. Without loss of generality, we focus on a finite field either $\mathbb{F} = \mathbb{Z}_p$ for a prime p or $\mathbb{F} = \mathbb{F}_{2^\lambda}$. We define a "gadget" vector $\boldsymbol{g} = (1, g, \ldots, g^{m-1})$ for $m = \lceil \log |\mathbb{F}| \rceil$, where $g = 2$ if $\mathbb{F} = \mathbb{Z}_p$ for a prime p and $g = X$ if $\mathbb{F} = \mathbb{F}_{2^\lambda}$. For a vector $\boldsymbol{x} \in \{0,1\}^m$, we have $\langle \boldsymbol{g}, \boldsymbol{x} \rangle = \sum_{i=1}^m x_i \cdot g^{i-1} \in \mathbb{F}$. We also denote by $\boldsymbol{g}^{-1} : \mathbb{F} \to \{0,1\}^m$ the bit-decomposition function that maps a field element $x \in \mathbb{F}$ to a bit vector $\boldsymbol{x} \in \{0,1\}^m$, such that $\langle \boldsymbol{g}, \boldsymbol{g}^{-1}(x) \rangle = x$. Following previous work (e.g., [BCG+20]), we allow a corrupted party to choose its output. If a sender $\mathcal{P}$ is corrupted, then it can introduce an error vector $\boldsymbol{e}$ into functionality $\mathcal{F}_{\mathsf{OLEe}}$. Then, $\mathcal{F}_{\mathsf{OLEe}}$ computes an error e' relying on the input y of a receiver $\mathcal{V}$. Finally, the error e' is added into the output z_2 of $\mathcal{V}$. The introduction of errors is asymmetric, i.e., $\mathcal{V}$ is *not*

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 7: **Functionality for homomorphic commitments.**

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

This functionality has all the features of $\mathcal{F}_{\mathsf{GP2PC}}$ (shown in Figure 11), 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 $(\mathsf{check}, \mathsf{id}_1, \ldots, \mathsf{id}_\ell)$ from $\mathcal{P}$ and $\mathcal{V}$, if $(\mathsf{id}_i, [\![y_i]\!])$ for all $i \in [1, \ell]$ were previously stored, send true to $\mathcal{V}$ if $y_i = 0$ for all $i \in [1, \ell]$ 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 8: **Functionality for IZK proofs based on IT-MACs.**

allowed to add an error into the output of $\mathcal{P}$. 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 7. 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 generic group

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, 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). The signature scheme is assumed to satisfy Existential UnForgeability under Chosen-Message Attack (EUF-CMA).
  - $(\mathbb{G}, p, q, G)$ 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. $\mathcal{C}$ interacts with $\mathcal{S}$ to generate a pair of session keys.

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. Client $\mathcal{C}$ 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 $(\mathsf{st}_C, \mathsf{st}_S) := (IV_C, IV_S)$, and run $\textsc{fin}_C \leftarrow \mathsf{stE.Enc}(\mathsf{key}_C, \ell_C, \textsc{h}_C, \textsc{ufin}_C, \mathsf{st}_C)$. Then, send $(\textsc{h}_C, \textsc{fin}_C)$ to $\mathcal{S}$.

   $\mathcal{S}$ initializes $(\mathsf{st}_C, \mathsf{st}_S) := (IV_C, IV_S)$, and runs $\textsc{ufin}_C \leftarrow \mathsf{stE.Dec}(\mathsf{key}_C, \textsc{h}_C, \textsc{fin}_C, \mathsf{st}_C)$. 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 $\textsc{fin}_S \leftarrow \mathsf{stE.Enc}(\mathsf{key}_S, \ell_S, \textsc{h}_S, \textsc{ufin}_S, \mathsf{st}_S)$, and then send $(\textsc{h}_S, \textsc{fin}_S)$ to $\mathcal{C}$.

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

Figure 9: **Handshake and record protocols for TLS 1.2.**

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 7 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,

Protocol TLS 1.2, continued

Record protocol execution with one-round session. $\mathcal{C}$ interacts with $\mathcal{S}$ to retrieve its personal data as follows:

6. *Client query:* Let 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 $\mathrm{ENC}_Q \leftarrow \mathsf{stE.Enc}(\mathsf{key}_C, \ell_Q, \mathrm{H}_Q, Q, \mathsf{st}_C)$, and sends $(\mathrm{H}_Q, \mathrm{ENC}_Q)$ to $\mathcal{S}$.
   (b) $\mathcal{S}$ runs $Q \leftarrow \mathsf{stE.Dec}(\mathsf{key}_C, \mathrm{H}_Q, \mathrm{ENC}_Q, \mathsf{st}_C)$, and generates R according to Q.
7. *Server response:* Let 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 $\mathrm{ENC}_R \leftarrow \mathsf{stE.Enc}(\mathsf{key}_S, \ell_R, \mathrm{H}_R, R, \mathsf{st}_S)$, and then sends $(\mathrm{H}_R, \mathrm{ENC}_R)$ to $\mathcal{S}$.
   (b) $\mathcal{C}$ gets $R \leftarrow \mathsf{stE.Dec}(\mathsf{key}_S, \mathrm{H}_R, \mathrm{ENC}_R, \mathsf{st}_d^C)$.

Figure 10: **Handshake and record protocols for TLS 1.2, contined.**

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 x is denoted by $[x]_p$ for former and $[x]_{2^{128}}$ for latter.

A.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 ZK 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 *local* 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]).

A.4 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 8, 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 $\mathcal{F}_{\mathsf{IZK}}$. Then, a prover $\mathcal{P}$ and the verifier $\mathcal{V}$ can authenticate witnesses via the (authinput) 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 $y - y' = 0$. Using known techniques [NNOB12, WYKW21], the IT-MACs can be checked in a batch with constant small communication. Therefore, we allow the (check) command to be called for a batch of IT-MACs.

B Secure 2PC Protocol via Garble-Then-Prove

In this section, we first present the formal definition of ideal functionality for 2PC via garble-then-prove. Then, we describe an efficient protocol to securely realize the ideal functionality, based on a plain Garbled-Circuit-based Two-Party Computation (GC-2PC) protocol with malicious OT and a ZK protocol based on IT-MACs. In Section **??**, we give the definition of garbling schemes.

B.1 Functionality for Garble-Then-Prove

In Figure 11, we give the detailed definition of ideal functionality $\mathcal{F}_{\mathsf{GP2PC}}$ in the garble-then-prove framework. In particular, the input, eval, output commands model the security of a plain Garbled-Circuit-based secure Two-Party Computation (GC-2PC) protocol, and the commit, prove commands abstract 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). Then, $\mathcal{P}$ can convince $\mathcal{V}$ that all 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 Section 4, we have shown that $\mathcal{F}_{\mathsf{GP2PC}}$ can 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. Through the (commit) command, the prover $\mathcal{P}$ is able to commit to bits, and these bits committed would in turn be used for proving the correctness of circuit evaluations. To open some bits committed, $\mathcal{P}$ and $\mathcal{V}$ can call the (output) command to let $\mathcal{V}$ obtain the bits. The bits provided by a malicious $\mathcal{P}$ for the (input) command may be inconsistent with that for the (commit) command. This is harmless for security, as $\mathcal{P}$'s bits were always committed before the reveal-and-prove phase. Through the

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).

Commit. Upon receiving $(\mathsf{commit}, \mathsf{id}, x)$ from $\mathcal{P}$ and $(\mathsf{commit}, \mathsf{id})$ from $\mathcal{V}$, where id is a fresh identifier and $x \in \{0,1\}$, store $(\mathsf{com}, \mathsf{id}, x)$ and send $(\mathsf{committed}, \mathsf{id})$ to both parties.

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 $\mathcal{P}$ and $\mathcal{V}$, where $A \in \{\mathcal{P}, \mathcal{V}, \text{both}\}$, and (id, z) or $(\mathsf{com}, \mathsf{id}, z)$ was previously stored, do the following:

1. If $A \in \{\mathcal{V}, \text{both}\}$, update (id, z) as $(\mathsf{id}, z, \mathit{output}, \mathcal{V})$ and output $(\mathsf{output}, \mathsf{id}, z)$ to $\mathcal{V}$.
2. If $A \in \{\mathcal{P}, \text{both}\}$, receive $e \in \{0,1\}$ from the adversary if $\mathcal{V}$ is corrupted, or set $e = 0$ otherwise.
3. If $A \in \{\mathcal{P}, \text{both}\}$, output $(\mathsf{output}, \mathsf{id}, z \oplus e)$ to $\mathcal{P}$.

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

1. Receive $(\mathsf{prove}, j, g_j, \mathbf{id}_j, \mathbf{id}'_j)$ from $\mathcal{P}$ and $\mathcal{V}$, where g_j is the verification circuit corresponding to the evaluation circuit f_j, either $\mathbf{id}_j = \bot$ or $(\mathsf{com}, \mathsf{id}_{j,i})$ for all $i \in [1,m]$ are present in memory, and $\mathbf{id}'_j$ is the vector of identifiers on the output $\boldsymbol{z}_j$ in the j-th (eval) call.
2. If $\mathbf{id}_j = \bot$, set $\boldsymbol{x}_j = \bot$. Otherwise, retrieve $(\mathsf{com}, \mathsf{id}_{j,i}, x_{j,i})$ for $i \in [1,m]$ and set $\boldsymbol{x}_j = (x_{j,1}, \ldots, x_{j,m})$. 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 (false) to $\mathcal{V}$. Otherwise, send (true) to $\mathcal{V}$.

Figure 11: **Reactive functionality for 2PC in the garble-then-prove framework.**

(output) command, $\mathcal{P}$, or $\mathcal{V}$, or both of them can obtain an output. For the case that both parties obtain the same output, we always consider that $\mathcal{V}$ first obtains the output and then $\mathcal{P}$ gets it.

Functionality $\mathcal{F}_{\mathsf{GP2PC}}$ 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 $\boldsymbol{x}_j$ 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 $\boldsymbol{x}_j$. 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 values outputted to it by recomputing these values. If the check fails, $\mathcal{P}$ could send abort to functionality $\mathcal{F}_{\mathsf{GP2PC}}$.

The (prove) 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

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. 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 the input $\boldsymbol{x}_j$ to generate a vector of IT-MACs $[\![\boldsymbol{x}_j]\!]$.
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 and opening of committed bits:* $\mathcal{P}$ and $\mathcal{V}$ execute the following steps.
   (a) If $\mathcal{P}$ will open a vector $\boldsymbol{v}_j$ that consists of a part of bits committed in IT-MACs, $\mathcal{P}$ sends $\boldsymbol{v}_j$ to $\mathcal{V}$, and then both parties call the (check) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on IT-MACs $[\![\boldsymbol{v}_j]\!] - \boldsymbol{v}_j$ to check that $\boldsymbol{v}_j$ is opened correctly.
   (b) If $\mathcal{V}$ will obtain the output $\boldsymbol{z}_j$, $\mathcal{P}$ sends D_j to $\mathcal{V}$ (where D_j can actually be sent in the preprocessing phase). Then $\mathcal{V}$ runs $\boldsymbol{z}_j \leftarrow \mathsf{Decode}(\mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j]), \mathrm{D}_j)$.
   (c) If both parties will output $\boldsymbol{z}_j$, $\mathcal{V}$ directly sends $\boldsymbol{z}_j$ (as computed in the previous step) to $\mathcal{P}$.
   (d) If only $\mathcal{P}$ will get $\boldsymbol{z}_j$, $\mathcal{V}$ sends $\mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j])$ to $\mathcal{P}$ who runs $\boldsymbol{z}_j \leftarrow \mathsf{Decode}(\mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j]), \mathrm{D}_j)$.

Online reveal-and-prove phase. In this 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 IT-MACs $[\![\mathsf{state}_{j-1}^*]\!]$, $[\![\boldsymbol{x}_j]\!]$ 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 $\boldsymbol{z}_j$, both parties call the (check) command of $\mathcal{F}_{\mathsf{IZK}}$ on IT-MACs $[\![\boldsymbol{z}_j^*]\!] - \boldsymbol{z}_j$ to check that $\boldsymbol{z}_j = \boldsymbol{z}_j^*$.

If any check fails, $\mathcal{V}$ outputs false. Otherwise, it outputs true.

Figure 12: **Protocol for securely instantiating $\mathcal{F}_{\mathsf{GP2PC}}$ in the $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{IZK}})$-hybrid model.**

$\mathcal{P}$ 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 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. Before the (revealandprove) command is called, the bits output to $\mathcal{P}$ or $\mathcal{V}$ may be incorrect. After the (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 also checked. That is, $\mathcal{F}_{\mathsf{GP2PC}}$ 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 $\boldsymbol{x}, \boldsymbol{y}$ of $\mathcal{P}$ and $\mathcal{V}$, a malicious $\mathcal{P}$ could learn $f'(\boldsymbol{x}', \boldsymbol{y}) = f(\boldsymbol{x}, \boldsymbol{y})$ in the reveal-and-prove phase for the malicious chosen circuit f' and input $\boldsymbol{x}'$. This leaks at most one-bit information on the input $\boldsymbol{y}$ of honest party $\mathcal{V}$. This is harmless as $\mathcal{P}$ could always get $\boldsymbol{y}$ in the reveal-and-prove phase. If $\mathcal{P}$ is honest, a malicious $\mathcal{V}$ cannot learn any secret information.

For applications that need to prevent $\mathcal{V}$ to introduce errors in the outputs (e.g., writing user data in the TLS server), we define an ideal functionality $\mathcal{F}^{\mathsf{noerr}}_{\mathsf{GP2PC}}$. This functionality is the same sas $\mathcal{F}_{\mathsf{GP2PC}}$, except that the adversary who corrupts $\mathcal{V}$ is *not* allowed to introduce any error for all (output) commands.

B.2 Instantiation of Functionality $\mathcal{F}_{\mathsf{GP2PC}}$

In Figure 12, we present a concretely efficient two-party protocol to instantiate functionality $\mathcal{F}_{\mathsf{GP2PC}}$ (shown in Figure 11) in the $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{IZK}})$-hybrid model based on a garbling scheme (defined in Section **??**). [6] This protocol combines a plain GC-2PC protocol with the recent interactive ZK proof modeled in functionality $\mathcal{F}_{\mathsf{IZK}}$. By calling functionality $\mathcal{F}_{\mathsf{IZK}}$, $\mathcal{P}$ could open a part of bits committed in IT-MACs to $\mathcal{V}$. For the sake of simplicity, we do not split a circuit 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 Figure 12, if an input vector $\boldsymbol{x}_j = \perp$ or $\boldsymbol{y}_j = \perp$, then the operations on the vector are ignored.

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 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.

The security of protocol Π_{GP2PC} (Figure 12) is stated in the following theorem. The detailed proof is postponed to Section **??**.

Theorem 2. *If the garbling scheme satisfies the simulation-based privacy and obliviousness, then protocol Π_{GP2PC} (shown in Figure 12) securely realizes functionality $\mathcal{F}_{\mathsf{GP2PC}}$ (shown in Figure 11) in the $(\mathcal{F}_{\mathsf{OT}}, \mathcal{F}_{\mathsf{IZK}})$-hybrid model, assuming H is a random oracle.*

In our protocol Π_{AuthData} shown in Section 4.1, a part of entries (e.g., initial vectors) in an input vector $\boldsymbol{x}_j$ could also be known by $\mathcal{V}$. In this case, $\boldsymbol{x}_j$ can be hardwired in the circuit g_j without needing to authenticate $\boldsymbol{x}_j$ with IT-MACs.

[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).

Extending the protocol to instantiate $\mathcal{F}_{\mathsf{GP2PC}}^{\mathsf{noerr}}$. If both parties will output $\boldsymbol{z}_j$, then $\mathcal{P}$ receives $\boldsymbol{z}_j$ from $\mathcal{V}$, and can compute $\mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j]) := \boldsymbol{z}_j \oplus \mathsf{lsb}(\mathsf{L}[\mathbf{0}_j])$ by itself. This case is equivalent to the case in which only $\mathcal{P}$ will get $\boldsymbol{z}_j$, and $\mathcal{P}$ directly receives $\mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j])$ from $\mathcal{V}$. Therefore, we only need to focus on the case that only $\mathcal{P}$ will get $\boldsymbol{z}_j$, when extending the protocol to securely realize functionality $\mathcal{F}_{\mathsf{GP2PC}}^{\mathsf{noerr}}$.

In protocol $\mathcal{F}_{\mathsf{GP2PC}}$ shown in Figure 12, the only way, which $\mathcal{V}$ introduces an error to an output, is to send a vector of bits $\boldsymbol{b}_j \neq \mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j])$ to $\mathcal{P}$. Thus, to guarantee that $\mathcal{P}$ obtains a correct output, we only need to add an authentication mechanism w.r.t. $\mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j])$ into the protocol $\mathcal{F}_{\mathsf{GP2PC}}$. The resulting protocol is natural to securely realize functionality $\mathcal{F}_{\mathsf{GP2PC}}^{\mathsf{noerr}}$. Specifically, let $S \subseteq [1, \ell]$ be the set of indices that $\mathcal{P}$ will obtain the outputs, i.e., for each $j \in S$, $\mathcal{P}$ will get $\boldsymbol{z}_j$. For each $j \in S$, except for sending $\mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j])$ to $\mathcal{P}$ who runs $\boldsymbol{z}'_j \leftarrow \mathsf{Decode}(\mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j]), \mathrm{D}_j)$, $\mathcal{V}$ computes $\mu := \mathsf{H}((\mathsf{L}[\boldsymbol{z}_j])_{j\in S})$, and $\mathcal{P}$ computes a vector of garbled labels $\mathsf{L}[\boldsymbol{z}'_j]$ and sets $\mu' := \mathsf{H}((\mathsf{L}[\boldsymbol{z}'_j])_{j\in S})$. Then, $\mathcal{P}$ and $\mathcal{V}$ run an equality-testing protocol, e.g., [HKE12], to verify that $\mu = \mu'$. If $\mu \neq \mu'$, then $\mathcal{P}$ aborts. A malicious $\mathcal{V}$ cannot forge garbled labels $\mathsf{L}[\boldsymbol{b}_j]$ with $\boldsymbol{b}_j \neq \mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j])$, and otherwise it will break the security of garbling scheme. Therefore, in the random oracle model, the probability that $\mu = \mu'$ but $\boldsymbol{b}_j \neq \mathsf{lsb}(\mathsf{L}[\boldsymbol{z}_j])$ is negligible in λ. The authentication mechanism only requires a small communication overhead, compared to the communication cost of protocol Π_{GP2PC}.

C Details of Our Authenticating-Data Protocol and Sub-Protocols for TLS Building Blocks

In this section, we first describe three sub-protocols that are used in our protocol Π_{AuthData}. Then we provide the details of protocol Π_{AuthData} to authenticate web data in TLS 1.2. In the handshake and record phases of protocol Π_{AuthData}, all secrets and application keys are stored by functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as its state, and then they are implicitly input by both parties to the sub-protocol executions. It is natural to see how state and state^* maintained by $\mathcal{F}_{\mathsf{GP2PC}}$ are updated from the protocol description.

In these protocols, we assume that functionality $\mathcal{F}_{\mathsf{IZK}}$ 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.

C.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.

Protocol Π_{E2F}

Inputs. $\mathcal{P}$ holds an elliptic-curve (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.**

C.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 4.1, 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 UFIN_C and UFIN_S and to check the correctness of UFIN_C and UFIN_S.
- 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

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 $IV_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret} \oplus \mathrm{ipad})$ as well as $IV_2 = f_{\mathsf{H}}(IV_0, \mathsf{secret} \oplus \mathrm{opad})$, 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 $\mathcal{F}_{\mathsf{GP2PC}}$ on the 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 ($\mathsf{eval}$) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on the 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. In the post-record phase of main protocol Π_{AuthData} (Figures 17 and 18), $\mathcal{P}$ performs the local verification to check the correctness of all values opened. That is, $\mathcal{P}$ performs the following checks involved in the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$. 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 the state secret^* and circuit $\mathcal{C}_1$ to generate $IV_2^* = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{opad})$ and check $IV_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{ipad})$.
6. For $i \in [1, n]$, both parties call the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on the 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 functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on the 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.**

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. It is easy to see how state and state^* maintained by $\mathcal{F}_{\mathsf{GP2PC}}$ are updated from the description of sub-protocol Π_{PRF}, and thus the state update of $\mathcal{F}_{\mathsf{GP2PC}}$ is omitted.

C.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 the stateful AEAD scheme based on AES-GCM. Protocol Π_{AEAD} works in the $(\mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{ZK}})$-hybrid model, and uses $\mathcal{F}_{\mathsf{GP2PC}}$ to compute AES blocks and prove their

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 $\mathsf{type}_1 =$ "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 $\mathsf{type}_1 =$ "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 $\mathsf{type}_2 =$ "open", $\mathcal{V}$ inputs the same plaintext $\mathbf{M}$ for $\mathsf{type}_1 =$ "encryption", and $n = 1$. Let $m \geq n+2$ be the maximum number of AES blocks used to generate any AEAD ciphertext in the TLS protocol execution. If $\mathsf{type}_2 =$ "secret", both parties input $\left[h^i\right]_{2^{128}}$ for all $i \in [1, m]$ where $h = \mathsf{AES}(\mathsf{key}, \mathbf{0})$.

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. Depending on the value of type_2, $\mathcal{P}$ and $\mathcal{V}$ do the following:
   - If $\mathsf{type}_2 =$ "open", then $\mathcal{P}$ samples $h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}} \leftarrow \{0,1\}^{128}$, and $\mathcal{P}$ and $\mathcal{V}$ call the (commit) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on the $\mathcal{P}$'s input $(h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}})$ to let $\mathcal{F}_{\mathsf{GP2PC}}$ commit to $(h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}})$.
   - If $\mathsf{type}_2 =$ "secret", then for each $i \in [0, n]$, $\mathcal{P}$ samples $\mathrm{Z}_{i,\mathcal{P}} \leftarrow \{0,1\}^{128}$, and both parties call the (commit) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on the $\mathcal{P}$'s input $\mathrm{Z}_{i,\mathcal{P}}$ to have $\mathcal{F}_{\mathsf{GP2PC}}$ commit to $\mathrm{Z}_{i,\mathcal{P}}$.

Handshake/record phase. In the handshake phase, functionality $\mathcal{F}_{\mathsf{GP2PC}}$ stores $\mathsf{key} \in \{\mathsf{key}_C, \mathsf{key}_S\}$.

2. 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 $\mathsf{type}_2 =$ "open", $\mathcal{P}$ and $\mathcal{V}$ call the (eval) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on the 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 $\mathsf{type}_2 =$ "secret", $\mathcal{P}$ and $\mathcal{V}$ call the (eval) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on the state key, Boolean 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 $i \in [0, n]$.
3. Depending on the values of type_1 and type_2, $\mathcal{P}$ and $\mathcal{V}$ do the following:
   - If $\mathsf{type}_2 =$ "open", then both parties compute $\mathrm{C}_1 := \mathrm{Z}_1 \oplus \mathrm{M}_1$ and set $\mathbf{C} := \mathrm{C}_1$ if $\mathsf{type}_1 =$ "encryption", or $\mathrm{M}_1 := \mathrm{Z}_1 \oplus \mathrm{C}_1$ and set $\mathbf{M} := \mathrm{M}_1$ if $\mathsf{type}_1 =$ "decryption".
   - If $\mathsf{type}_2 =$ "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 $\mathrm{C}_i := \mathrm{B}_i \oplus \mathrm{Z}_{i,\mathcal{V}}$ if $\mathsf{type}_1 =$ "encryption". For each $i \in [1, n]$, $\mathcal{V}$ sends $\mathrm{Z}_{i,\mathcal{V}}$ to $\mathcal{P}$, who sets $\mathrm{C}_i := \mathrm{B}_i \oplus \mathrm{Z}_{i,\mathcal{V}}$ if $\mathsf{type}_1 =$ "encryption", or $\mathrm{M}_i := \mathrm{Z}_{i,\mathcal{P}} \oplus \mathrm{Z}_{i,\mathcal{V}} \oplus \mathrm{C}_i$ if $\mathsf{type}_1 =$ "decryption". Both parties set $\mathbf{C} := (\mathrm{C}_1, \ldots, \mathrm{C}_n)$ if $\mathsf{type}_1 =$ "encryption", or $\mathcal{P}$ sets $\mathbf{M} := (\mathrm{M}_1, \ldots, \mathrm{M}_n)$ if $\mathsf{type}_1 =$ "decryption".

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

Protocol Π_{AEAD}, continued

4. If type_2 = "open", $\mathcal{P}$ and $\mathcal{V}$ define an additive sharing $[h]_{2^{128}} = (h_{\mathcal{P}}, h_{\mathcal{V}})$. Then, both parties generate a multiplication sharing $(\widetilde{h}_{\mathcal{P}}, \widetilde{h}_{\mathcal{V}})$ such that $h = \widetilde{h}_{\mathcal{P}} \cdot \widetilde{h}_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$ as follows:
   (a) $\mathcal{P}$ samples $\widetilde{h}_{\mathcal{P}} \leftarrow \mathbb{F}_{2^{128}} \backslash \{0\}$. Then, $\mathcal{P}$ (as a sender) and $\mathcal{V}$ (as a receiver) call functionality $\mathcal{F}_{\mathsf{OLEe}}$ on respective input $(\widetilde{h}_{\mathcal{P}})^{-1}$ and $h_{\mathcal{V}}$ to obtain an additive sharing $(s_{\mathcal{P}}, s_{\mathcal{V}})$ such that $s_{\mathcal{P}} + s_{\mathcal{V}} = (\widetilde{h}_{\mathcal{P}})^{-1} \cdot h_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$.
   (b) $\mathcal{P}$ computes $d := (\widetilde{h}_{\mathcal{P}})^{-1} \cdot h_{\mathcal{P}} + s_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$ and sends it to $\mathcal{V}$, who computes $\widetilde{h}_{\mathcal{V}} := d + s_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$. Thus, we have $\widetilde{h}_{\mathcal{P}} \cdot \widetilde{h}_{\mathcal{V}} = \widetilde{h}_{\mathcal{P}} \cdot ((\widetilde{h}_{\mathcal{P}})^{-1} \cdot h_{\mathcal{P}} + s_{\mathcal{P}} + s_{\mathcal{V}}) = \widetilde{h}_{\mathcal{P}} \cdot ((\widetilde{h}_{\mathcal{P}})^{-1} \cdot h_{\mathcal{P}} + (\widetilde{h}_{\mathcal{P}})^{-1} \cdot h_{\mathcal{V}}) = h_{\mathcal{P}} + h_{\mathcal{V}} = h$.
5. If type_2 = "open", from $i = 2$ to m, $\mathcal{P}$ and $\mathcal{V}$ compute an additive sharing $\left[h^i\right]_{2^{128}}$ by letting $\mathcal{P}$ (as a sender) and $\mathcal{V}$ (as a receiver) call functionality $\mathcal{F}_{\mathsf{OLEe}}$ on respective input $(\widetilde{h}_{\mathcal{P}})^i$ and $(\widetilde{h}_{\mathcal{V}})^i$ to obtain an additive sharing $\left[h^i\right]_{2^{128}} = (a_i, b_i)$ such that $a_i + b_i = (\widetilde{h}_{\mathcal{P}})^i \cdot (\widetilde{h}_{\mathcal{V}})^i = h^i \in \mathbb{F}_{2^{128}}$.
6. $\mathcal{P}$ and $\mathcal{V}$ define an additive sharing $[\mathrm{Z}_0]_{2^{128}} = (\mathrm{Z}_{0,\mathcal{P}}, \mathrm{Z}_{0,\mathcal{V}})$. For $\mathbf{C} = (\mathrm{C}_1, \ldots, \mathrm{C}_n)$, the parties compute an additive sharing $[\sigma]_{2^{128}} = (\sigma_{\mathcal{P}}, \sigma_{\mathcal{V}})$ of a GMAC tag $\sigma = \mathrm{Z}_0 \oplus \Phi_{(\mathrm{H}, \mathbf{C}, \ell_{\mathrm{H}}, \ell_{\mathrm{C}})}(h)$ as follows:
   - Let $w \in \mathbb{F}_{2^{128}}$ be the field element corresponding to the bit-vector $(\ell_{\mathrm{H}}, \ell_{\mathrm{C}})$.
   - Both parties locally compute $[\sigma]_{2^{128}} := [\mathrm{Z}_0]_{2^{128}} + \mathrm{H} \cdot \left[h^{n+2}\right]_{2^{128}} + \sum_{i=1}^{n} \mathrm{C}_i \cdot \left[h^{n+2-i}\right]_{2^{128}} + w \cdot [h]_{2^{128}}$.
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 $\mathrm{CT} = (\mathbf{C}, \sigma)$.
   - 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. The following step 8 was performed before the (revealandprove) command of $\mathcal{F}_{\mathsf{GP2PC}}$ is called. In this phase, $\mathcal{F}_{\mathsf{GP2PC}}$ stores key^* with $\mathsf{key}^* = \mathsf{key}$ in the honest case, and $\mathcal{P}$ knows key^*. In the post-record phase of main protocol Π_{AuthData} (Figures 17 and 18), $\mathcal{P}$ performs the local verification to check the correctness of each AEAD ciphertext. That is, $\mathcal{P}$ performs the following checks involved in the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$. In this phase, $\mathcal{P}$ and $\mathcal{V}$ execute the following steps:

8. If type_2 = "open", both parties call the (output) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ to open $(h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}})$ to $\mathcal{V}$, and then $\mathcal{V}$ locally 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 type_2 = "secret", both parties call the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ to open $\mathrm{Z}_{0,\mathcal{P}}$ to $\mathcal{V}$, and then $\mathcal{V}$ locally computes $\mathrm{Z}_0 := \mathrm{Z}_{0,\mathcal{P}} \oplus \mathrm{Z}_{0,\mathcal{V}}$.
9. If type_2 = "open", both parties call the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on the state key^* and circuit $\mathcal{C}'_{aes}$ to check $h = \mathsf{AES}(\mathsf{key}^*, \mathbf{0})$, $\mathrm{Z}_0 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}_0)$, and $\mathrm{Z}_1 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}_1)$. If type_2 = "secret", both parties call the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on the state key^*, committed inputs $(\mathrm{Z}_{1,\mathcal{P}}, \ldots, \mathrm{Z}_{n,\mathcal{P}})$ and circuit $\mathcal{D}'_{aes}$ to check $\mathrm{Z}_0 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}_0)$ and $\mathrm{Z}_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}_i) \oplus \mathrm{Z}_{i,\mathcal{P}}$ for $i \in [1, n]$. In both cases, $\mathcal{V}$ aborts if the check fails.
10. If type_2 = "secret", for $i \in [1, n]$, $\mathcal{P}$ and $\mathcal{V}$ call the (authinput) command of $\mathcal{F}_{\mathsf{IZK}}$ on $\mathrm{Z}_{i,\mathcal{P}}$ to generate $[\![\mathrm{Z}_{i,\mathcal{P}}]\!]$. In this case, both parties locally compute $[\![\mathrm{Z}_i]\!] := [\![\mathrm{Z}_{i,\mathcal{P}}]\!] \oplus \mathrm{Z}_{i,\mathcal{V}}$ for $i \in [1, n]$. Then, $\mathcal{P}$ and $\mathcal{V}$ locally compute $[\![\mathrm{M}_i]\!] := [\![\mathrm{Z}_i]\!] \oplus \mathrm{C}_i$ for $i \in [1, n]$, and set $[\![\mathbf{M}]\!] = ([\![\mathrm{M}_1]\!], \ldots, [\![\mathrm{M}_n]\!])$.

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

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). Functionality $\mathcal{F}_{\mathsf{IZK}}$ is used for generating IT-MACs on the plaintexts underlying the AEAD ciphertexts. Recall that $\mathcal{F}_{\mathsf{IZK}}$ inherits the commands of $\mathcal{F}_{\mathsf{GP2PC}}$, and thus can directly authenticate the inputs and outputs of $\mathcal{F}_{\mathsf{GP2PC}}$ with IT-MACs. To support the extension of multi-round sessions of the main protocol Π_{AuthData} (described in Section 4.2), sub-protocol Π_{AEAD} is designed to cover the case of decrypting

AEAD ciphertexts in the record phase. 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 one-bit information of h is revealed.

When applying sub-protocol Π_{AEAD} into our protocol Π_{AuthData} (shown in Section 4.1), 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 be 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", Π_{AEAD} is used to generate a client finished message FIN_C.
- If type_1 = "decryption" and type_2 = "open", Π_{AEAD} is used to generate a server finished message FIN_S. In this case, an optimization is described in Section 4.2.
- If type_1 = "encryption" and type_2 = "secret", Π_{AEAD} is used to generate AEAD ciphertexts and IT-MACs on queries.
- If type_1 = "decryption" and type_2 = "secret", Π_{AEAD} is used to decrypt and verify AEAD ciphertexts of responses and to generate IT-MACs on 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 $\mathcal{V}$, both parties can execute sub-protocol Π_{AEAD} to decrypt every AEAD ciphertext from the server (see Section 4.2 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 Π_{AEAD} and reused to generate GMAC tags.

When applying OLE with errors and additive secret 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 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. 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.4 Details of Protocol Π_{AuthData}

In Figures 17 and 18, we give the details of 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. This protocol invokes three sub-protocols Π_{E2F}, Π_{PRF}, Π_{AEAD}, which have been described in the previous subsections. Protocol Π_{AuthData} 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 sub-protocols $\Pi_{\mathsf{E2F}}, \Pi_{\mathsf{AEAD}}$, 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 $\mathcal{F}_{\mathsf{GP2PC}}$ invoked by Π_{PRF} can be directly used in Π_{AEAD}. Since Π_{PRF} and Π_{AEAD} are invoked multiple times, and we use $\Pi^{(i)}_{\mathsf{PRF}}$

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 Figures 9 and 10. $\mathcal{P}$ holds a private input α for a query template Query known by both parties. Let F_x be a function mapping an elliptic-curve point to its x-coordinate.

- **Preprocessing : Generate correlated randomness.** Before starting a TLS session, $\mathcal{P}$ and $\mathcal{V}$ run the preprocessing phase of sub-protocols Π_{E2F} (Figure 13) and Π_{AEAD} (Figure 15) to generate random additive sharings. Both parties call functionality $\mathcal{F}_{\mathsf{IZK}}$ (Figure 8) to initialize a global key $\Delta \in \{0,1\}^\lambda$ sampled uniformly by $\mathcal{V}$.
- **Handshake : Generate pre-master secret.**
  1. $\mathcal{P}$ samples $r_C \leftarrow \{0,1\}^{256}$ and sends $\text{REQ}_C := r_C$ to $\mathcal{S}$. Then, $\mathcal{P}$ receives $\text{RES}_S = (r_S, T_S, \mathsf{cert}_S, \sigma_S)$ from $\mathcal{S}$, and sends $(\text{REQ}_C, \text{RES}_S)$ 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 $\text{RES}_C := T_C$ 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} (Figure 13) on respective inputs 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, $\mathcal{P}$ and $\mathcal{V}$ call the (input) and (eval) commands of $\mathcal{F}_{\mathsf{GP2PC}}$ on respective inputs $(\mathsf{pms}_\mathcal{P}, \mathsf{pms}_\mathcal{V})$ and common circuit *AddModp* to generate and store pms. Both parties also call the (commit) command of $\mathcal{F}_{\mathsf{GP2PC}}$ on the $\mathcal{P}$'s input $\mathsf{pms}_\mathcal{P}$ to commit to $\mathsf{pms}_\mathcal{P}$.
  5. Both parties run sub-protocol $\Pi^{(1)}_{\mathsf{PRF}}$ (Figure 14) on the state pms, message $(\text{“master secret”}, r_C \| r_S)$ and a label “secret” to make $\mathcal{F}_{\mathsf{GP2PC}}$ generate and store $\mathsf{ms} = \mathsf{PRF}_{384}(\mathsf{pms}, \text{“master secret”}, r_C \| r_S)$.
  6. $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi^{(2)}_{\mathsf{PRF}}$ (Figure 14) on the state ms, message $(\text{“key expansion”}, r_S \| r_C)$ 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 $(\mathsf{st}_C, \mathsf{st}_S) := (IV_C, IV_S)$.
- **Handshake : Exchange finished messages.**
  7. $\mathcal{P}$ and $\mathcal{V}$ compute $\tau_C := \mathsf{H}(\text{REQ}_C \| \text{RES}_S \| \text{RES}_C)$. Then, both parties run sub-protocol $\Pi^{(3)}_{\mathsf{PRF}}$ (Figure 14) on the state ms, message $(\text{“client finished”}, \tau_C)$ and a label “open” to let the parties obtain $\text{UFIN}_C = \mathsf{PRF}_{96}(\mathsf{ms}, \text{“client finished”}, \tau_C)$.
  8. $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi^{(1)}_{\mathsf{AEAD}}$ (Figure 15) on the state key_C, a tuple $(\mathsf{st}_C, \ell_C, \text{H}_C, \text{UFIN}_C)$, a label “encryption” and a label “open” to generate $\text{FIN}_C = \mathsf{stE.Enc}(\mathsf{key}_C, \ell_C, \text{H}_C, \text{UFIN}_C, \mathsf{st}_C)$. During this execution, both parties obtain $\left[h^i_C\right]_{2^{128}}$ for $i \in [1, m]$ where $h_C = \mathsf{AES}(\mathsf{key}_C, \mathbf{0})$ and $m = \lceil |Q|/128 \rceil + 2$. Both parties update $\mathsf{st}_C := \mathsf{st}_C + 2$. $\mathcal{P}$ sends $(\text{H}_C, \text{FIN}_C)$ to $\mathcal{S}$, who checks correctness of FIN_C and UFIN_C following the TLS specification and aborts if the check fails.
  9. After receiving $(\text{H}_S, \text{FIN}_S)$ from $\mathcal{S}$, $\mathcal{P}$ forwards it to $\mathcal{V}$. Then, $\mathcal{P}$ and $\mathcal{V}$ compute $\tau_S := \mathsf{H}(\text{REQ}_C \| \text{RES}_S \| \text{RES}_C \| \text{UFIN}_C)$. Both parties execute sub-protocol $\Pi^{(4)}_{\mathsf{PRF}}$ (Figure 14) on the state ms, message $(\text{“server finished”}, \tau_S)$ and label “open” such that both parties get $\text{UFIN}_S = \mathsf{PRF}_{96}(\mathsf{ms}, \text{“server finished”}, \tau_S)$.
  10. $\mathcal{P}$ and $\mathcal{V}$ run sub-protocol $\Pi^{(2)}_{\mathsf{AEAD}}$ (Figure 15) on the state key_S, a tuple $(\mathsf{st}_S, \ell_S, \text{H}_S, \text{FIN}_S)$, a label “decryption” and a label “open” such that both parties get UFIN_S'. Then, $\mathcal{P}$ and $\mathcal{V}$ check that $\text{UFIN}_S = \text{UFIN}_S'$ and abort if the check fails. The parties update $\mathsf{st}_S := \mathsf{st}_S + 2$.

Figure 17: **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 $\Pi^{(3)}_{\mathsf{AEAD}}$ (Figure 15) on the state key_C, a tuple $\big(\mathsf{st}_C, \ell_Q, \mathrm{H}_Q, Q, \{[h^i_C]_{2^{128}}\}_{i\in[1,m]}\big)$, a label "encryption" and a label "secret" such that both parties obtain $\mathrm{ENC}_Q = \mathsf{stE.Enc}(\mathsf{key}_C, \ell_Q, \mathrm{H}_Q, Q, \mathsf{st}_C)$. Then, $\mathcal{P}$ sends $(\mathrm{H}_Q, \mathrm{ENC}_Q)$ to $\mathcal{S}$.
12. Following the TLS specification, $\mathcal{S}$ checks the correctness of $(\mathrm{H}_Q, \mathrm{ENC}_Q)$, and aborts if the check fails. Then $\mathcal{S}$ decrypts ENC_Q to get Q, and computes a ciphertext $(\mathrm{H}_R, \mathrm{ENC}_R)$ on a response R. Then, $\mathcal{S}$ sends $(\mathrm{H}_R, \mathrm{ENC}_R)$ 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 $(h_C, h_S, \mathrm{z}_C, \mathrm{z}_S, \mathrm{z}_Q)$, where $\mathrm{z}_C, \mathrm{z}_S, \mathrm{z}_Q$ are the AES blocks used to generate the GMAC tags involved in $\mathrm{FIN}_C, \mathrm{FIN}_S, \mathrm{ENC}_Q$.
14. $\mathcal{P}$ and $\mathcal{V}$ call the (revealandprove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$, which sends $\mathsf{pms}_{\mathcal{V}}$ to $\mathcal{P}$. 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 $\mathsf{pms}^* = \mathit{AddModp}(\mathsf{pms}_{\mathcal{P}}, \mathsf{pms}_{\mathcal{V}})$.
    (b) $\mathcal{P}$ uses pms^* to check the correctness of all values opened during sub-protocol execution $\Pi^{(1)}_{\mathsf{PRF}}$, 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 $\mathrm{UFIN}_C, \mathrm{UFIN}_S$ are checked.
    (d) $\mathcal{P}$ uses $(\mathsf{key}^*_C, \mathsf{key}^*_S, \mathrm{UFIN}_C, \mathrm{UFIN}_S, Q)$ to check the correctness of FIN_C, FIN_S and ENC_Q by recomputing these ciphertexts and comparing them with ciphertexts FIN_C, FIN_S and ENC_Q.
    (e) $\mathcal{P}$ runs $R \leftarrow \mathsf{stE.Dec}(\mathsf{key}^*_S, \mathrm{H}_R, \mathrm{ENC}_R, \mathsf{st}_S)$. If stE.Dec outputs $\perp$, $\mathcal{P}$ aborts.
15. Let $\overline{\mathit{AddModp}}$ be a Boolean circuit which inputs $\mathsf{pms}_{\mathcal{P}}$ and outputs $\mathsf{pms}^* = \mathit{AddModp}(\mathsf{pms}_{\mathcal{P}}, \mathsf{pms}_{\mathcal{V}})$ for a common value $\mathsf{pms}_{\mathcal{V}}$. $\mathcal{P}$ and $\mathcal{V}$ call the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ on the committed input $\mathsf{pms}_{\mathcal{P}}$ and circuit $\overline{\mathit{AddModp}}$ to generate and store pms^*.
16. Given pms^* stored in $\mathcal{F}_{\mathsf{GP2PC}}$, by calling functionalities $\mathcal{F}_{\mathsf{GP2PC}}$ and $\mathcal{F}_{\mathsf{IZK}}$, $\mathcal{P}$ 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. If the check fails, $\mathcal{V}$ aborts. During these sub-protocol executions, key^*_S was stored by $\mathcal{F}_{\mathsf{IZK}}$, and $[\![Q]\!]$ was generated.
17. Both parties call the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on the state key^*_S and Boolean circuit $\mathit{AES}[\mathsf{st}_S]$ to generate $[\![\mathrm{z}_R]\!]$, where $\mathit{AES}[\mathsf{st}_S]$ takes as input key^*_S and outputs $\mathrm{z}_R = \mathsf{AES}(\mathsf{key}^*_S, \mathsf{st}_S)$. $\mathcal{P}$ sends z_R to $\mathcal{V}$, and then both parties call the (check) command of $\mathcal{F}_{\mathsf{IZK}}$ on $[\![\mathrm{z}_R]\!] - \mathrm{z}_R$ to check that z_R received by $\mathcal{V}$ is correct. Then, $\mathcal{V}$ uses $(h_C, h_S, \mathrm{z}_C, \mathrm{z}_S, \mathrm{z}_Q, \mathrm{z}_R)$ to check the correctness of all GMAC tags in the AEAD ciphertexts $\mathrm{FIN}_C, \mathrm{FIN}_S, \mathrm{ENC}_Q, \mathrm{ENC}_R$ following the AEAD specification, and aborts if the check fails.
18. $\mathcal{P}$ and $\mathcal{V}$ parse $\mathrm{ENC}_R = (\mathbf{C}_R, \sigma_R)$. Both parties call the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on the state key^*_S and Boolean circuit $\mathit{AESDec}[\mathsf{st}_S, \mathbf{C}_R]$ to generate $[\![R]\!]$, where $\mathit{AESDec}[\mathsf{st}_S, \mathbf{C}_R]$ takes as input key^*_S, and decrypts $\mathbf{C}_R$ to an output R with key^*_S and st_S.

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

19. $\mathcal{P}$ and $\mathcal{V}$ call the (convert) command of functionality $\mathcal{F}_{\mathsf{Conv}}$ (Figure 19) on IT-MACs $([\![Q]\!], [\![R]\!])$ 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 18: **Protocol of authenticating web data for TLS, continued.**

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 8) and $\mathcal{F}_{\mathsf{HCom}}$ (shown in Figure 7). 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 19: **Functionality for converting IT-MACs to additively homomorphic commitments.**

(resp., $\Pi_{\mathsf{AEAD}}^{(i)}$) 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 $\mathcal{F}_{\mathsf{GP2PC}}$. When $\mathcal{F}_{\mathsf{GP2PC}}$ and $\mathcal{F}_{\mathsf{IZK}}$ are instantiated, garbled circuits and random IT-MACs are generated in the preprocessing phase.

D 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 7. In the main protocol Π_{AuthData} (shown in Section 4.1), we use a conversion functionality $\mathcal{F}_{\mathsf{Conv}}$ (shown in Figure 19) to convert IT-MACs into AHCs. The definition of the $(\mathsf{convert})$ command of functionality $\mathcal{F}_{\mathsf{Conv}}$ is 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 20, 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 $\mathbb{F}$. [7]

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 20), 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

[7] 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 $\mathcal{F}_{\mathsf{HCom}}[\mathbb{F}]$ 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 20: **Protocol for converting IT-MACs to AHCs in the $(\mathcal{F}_{\mathsf{Com}}, \mathcal{F}_{\mathsf{HCom}}, \mathcal{F}_{\mathsf{OLEe}})$-hybrid model.**

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 $\Delta' = \Delta \oplus E_1$ for an error $E_1 \neq 0$ chosen by $\mathcal{V}$. Then, $\mathcal{V}$ could open Δ' along with Γ to an honest prover $\mathcal{P}$ who uses $\mathsf{K}'[u_i] = \mathsf{M}[u_i] \oplus u_i\Delta \oplus u_iE_1 = \mathsf{K}[u_i] \oplus u_iE_1$ to check the correctness of W_i for $i \in [1,n]$, where $\mathsf{K}[u_i] =$

$\mathsf{M}[u_i] \oplus u_i\Delta$. For $i \in [1,n]$, malicious verifier $\mathcal{V}$ can construct $W_i = \mathsf{H}(\mathsf{K}[u_i]) - \mathsf{H}(\mathsf{K}[u_i] \oplus \Delta \oplus E_1) + \Gamma$ to guess $u_i = 0$ or $W_i = \mathsf{H}(\mathsf{K}[u_i] \oplus E_1) - \mathsf{H}(\mathsf{K}[u_i] \oplus \Delta) + \Gamma$ to guess $u_i = 1$. This allows $\mathcal{V}$ to perform a selective failure attack on some bits of $\boldsymbol{u}$. 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 (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 $\mathcal{V}$ 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. In addition, a malicious $\mathcal{V}$ may use an inconsistent $\Gamma' = \Gamma + E_2$ for an error $E_2 \neq 0$ chosen by $\mathcal{V}$ when calling functionality $\mathcal{F}_{\mathsf{OLEe}}$, which results in an incorrect correlation $\widetilde{\mathsf{M}}[u_0] = \widetilde{\mathsf{K}}[u_0] + u_0 \cdot \Gamma + u_0 \cdot E_2$. In this case, we have that $\widetilde{\mathsf{M}}[y] = \widetilde{\mathsf{K}}[y] + y \cdot \Gamma + u_0 \cdot E_2$, where $\widetilde{\mathsf{K}}[y] = \sum_{i\in[1,n]} \chi_i \cdot \widetilde{\mathsf{K}}[u_i] + \widetilde{\mathsf{K}}[u_0]$. After $(y, \widetilde{\mathsf{M}}[y])$ is opened to $\mathcal{V}$, it can compute $u_0 := (\widetilde{\mathsf{M}}[y] - \widetilde{\mathsf{K}}[y] - y \cdot \Gamma)/E_2$ and then reveals 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 $(y, \widetilde{\mathsf{M}}[y])$ is opened. If $\mathcal{V}$ sends $\widetilde{\mathsf{K}}[y]' = \widetilde{\mathsf{K}}[y] + E_3$ to $\mathcal{P}$, then we obtain that $u_0 \cdot E_2 + E_3 = 0$, which occurs with probability at most $1/|\mathbb{F}|$, as u_0 is uniform in $\mathbb{F}$ and (E_2, E_3) is independent of u_0. For more details, see the proof of Theorem 3.

We can extend protocol Π_{Conv} to securely realize functionality $\mathcal{F}_{\mathsf{Conv}}$ without the (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

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

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 3. *Protocol Π_{Conv} (shown in Figure 20) securely realizes functionality $\mathcal{F}_{\mathsf{Conv}}$ (shown in Figure 19) 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 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 ($\mathsf{open}$) 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 $\tau = \mathsf{H}'(\mathsf{M}[v_1] \oplus v_1\Delta \oplus v_1 E_1, \ldots, \mathsf{M}[v_\lambda] \oplus v_\lambda \Delta \oplus v_\lambda E_1) = \mathsf{H}'(\mathsf{K}[v_1] \oplus v_1 E_1, \ldots, \mathsf{K}[v_\lambda] \oplus v_\lambda E_1)$. If $\mathcal{A}$ makes a query $(\mathsf{K}[v_1] \oplus v_1 E_1, \ldots, \mathsf{K}[v_\lambda] \oplus v_\lambda E_1)$ 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 $\tau = \mathsf{H}'(\mathsf{K}[v_1] \oplus v_1 E_1, \ldots, \mathsf{K}[v_\lambda] \oplus v_\lambda E_1)$ 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 20), 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.

E Proof of Security for Π_{AuthData}

In this section, we give the detailed proof of Theorem 1. Before giving the proof, we recall the PRF-ODH assumption [JKSS12, KPW13] as follows.

Definition 1. *Let $\widehat{F}$ be a PRF that takes as input a key element from $\mathbb{G}$ and a message, and then outputs a string of length λ. We say that the PRF-ODH assumption holds for a function $\widehat{F}$ if all PPT adversaries $\mathcal{A}$, the $\mathcal{A}$'s advantage in the following experiment is negligible in λ.*

Let $(\mathbb{G}, q, G)$ be an EC group. Given $y \in \mathbb{Z}_q$, we define an oracle $\mathsf{ODH}_y(X, m)$ that outputs $\widehat{F}(y \cdot X, m)$, where $X \in \mathbb{G}$ and m is in the domain of $\widehat{F}$. The PRF-ODH experiment between a challenger and an adversary $\mathcal{A}$ is described as follows.

1. *The challenger samples $y \leftarrow \mathbb{Z}_q$ and computes $Y := y \cdot G$, and then sends Y to $\mathcal{A}$.*
2. *For each query (U, m) with $U \in \mathbb{G}$ from $\mathcal{A}$, the challenger returns $\mathsf{ODH}_y(U, m)$.*
3. *At some point, $\mathcal{A}$ sends m^* to the challenger. Then, the challenger samples $x \leftarrow \mathbb{Z}_q, z_1 \leftarrow \{0,1\}^\lambda, b \leftarrow \{0,1\}$, and then computes $X := x \cdot G$ and $z_0 := \widehat{F}(x \cdot Y, m^*)$. the challenger sends (X, z_b) to $\mathcal{A}$.*
4. *$\mathcal{A}$ keeps querying the oracle ODH_y via the challenger, except that it is not allowed to query the pair (X, m^*).*
5. *At the end, $\mathcal{A}$ outputs a bit b'. We say that $\mathcal{A}$ wins in the PRF-ODH experiment if $b' = b$.*

For proving the security of protocol Π_{AuthData}, we can restrict that $\mathcal{A}$ could make at most two queries to the ODH oracle after the pair (X, m^*) was sent. We can instantiate $\widehat{F}(K, m)$ as $f_{\mathsf{H}}\big(\mathsf{F}_x(K) \oplus m\big)$, which is a PRF in the Random Oracle Model (ROM), where F_x maps an EC point to its x-coordinate. In addition, for the provable security of protocol Π_{AuthData}, we need to provide adversary $\mathcal{A}$ with the following selective-failure queries:

- $\mathcal{A}$ sends a predicate $P : \mathbb{G} \times \mathbb{G} \to \{0,1\}$ to the challenger, who returns $P(X, x \cdot Y)$ to $\mathcal{A}$. If $P(X, x \cdot Y) = 0$, the challenger aborts and the experiment outputs a random bit b'.

We restrict that $\mathcal{A}$ can make the above selective-failure queries *at most three times*. Through querying a predicate, $\mathcal{A}$ could guess a few bits of X and $x \cdot Y$. However, an incorrect guess would

incur that the experiment aborts. Overall, $\mathcal{A}$ could learn (on average) one-bit information of X or $x \cdot Y$ via the selective-failure queries. Therefore, the PRF-ODH assumption still holds, even if adversary $\mathcal{A}$ is allowed to make the selective-failure queries three times.

Theorem 4 (Theorem 1, restated)**.** *If the PRF-ODH assumption holds and the underlying signature scheme is EUF-CMA secure, then protocol* Π_{AuthData} *(shown in Figures 17 and 18) 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 prover $\mathcal{P}$ and then consider the case of a malicious verifier $\mathcal{V}$. 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 the 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. For the construction of each sub-simulator, we also show the simulated view against adversary $\mathcal{A}$ is computationally indistinguishable from the real view. For a value in the proof, we use V to denote the actual value that should be computed and V^* to represent the corresponding value used in the ZK proof of the post-record phase. For the simulation of a sub-protocol, we use V' to denote the inconsistent value due to the malicious behavior of adversary $\mathcal{A}$, and denote by $e = V' - V$ the error chosen by $\mathcal{A}$. For the simulation of main protocol, we straightforwardly use $V \oplus \mathrm{E}$ to denote the inconsistent value for some error E.

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.

E.1 Proof for Malicious Prover

We first give the constructions of sub-simulators $\mathcal{S}_{\mathsf{E2F}}$, $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$ for three sub-protocols, and then describe the construction of simulator $\mathcal{S}$ 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 $\mathcal{F}_{\mathsf{AuthData}}$. For the security analysis of each sub-protocol, we always cast it to the main protocol Π_{AuthData} and then analyze its security.

Simulation and analysis of sub-protocol Π_{E2F} **(Figure 13).** 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 an input tuple $(a_1, b_1, a_1, b'_1, r_1)$, error vectors $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_5$ and $\mathcal{P}$'s shares of $[a_1 b_2]_p, [a_2 b_1]_p, [a_1 b'_2]_p, [a_2 b'_1]_p, [r_1 r_2]_p$. Then, $\mathcal{S}_{\mathsf{E2F}}$ samples $a_2, b_2, b'_2, r_2 \leftarrow \mathbb{Z}_p$, and computes $e_1 = \langle \boldsymbol{g} * \boldsymbol{e}_1, \boldsymbol{b}_2 \rangle$, $e_2 = \langle \boldsymbol{g} * \boldsymbol{e}_2, \boldsymbol{a}_2 \rangle$, $e_3 = \langle \boldsymbol{g} * \boldsymbol{e}_3, \boldsymbol{b}'_2 \rangle$, $e_4 =$

$\langle \boldsymbol{g} * \boldsymbol{e}_4, \boldsymbol{a}_2 \rangle$ and $e_5 = \langle \boldsymbol{g} * \boldsymbol{e}_5, \boldsymbol{r}_2 \rangle$, where $\boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{b}'_2, \boldsymbol{r}_2$ are the bit-decomposition of field elements $a_2, b_2, b'_2, r_2 \in \mathbb{Z}_p$. Following the definition of $\mathcal{F}_{\mathsf{OLEe}}$, $\mathcal{S}$ computes $\mathcal{V}$'s shares on $[a_1 b_2]_p, [a_2 b_1]_p$, $[a_1 b'_2]_p, [a_2 b'_1]_p, [r_1 r_2]_p$.

2. $\mathcal{S}_{\mathsf{E2F}}$ computes the shares of $[c]_p$, $[c']_p$ and $[r^2]_p$ held by $\mathcal{P}$ and $\mathcal{V}$ 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 $\epsilon_1, \epsilon_2, \epsilon_3, w$ following the protocol specification. Note that $\mathcal{S}_{\mathsf{E2F}}$ could know the point $Z_2 = (x_2, y_2)$ according to the following simulation of main protocol Π_{AuthData}. Then $\mathcal{S}_{\mathsf{E2F}}$ is able to compute $\mathcal{V}$'s shares on 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 $e_z = f(a, r, e_c, e_{c'}, e_r, e_6, e_7, e_8, e_9)$, which is added into the secret on $[z]_p$, where f is a function depending on the definition of z.

Let $[z]_p = (z_1, z_2)$ such that $z_1 + z_2 = z \mod p$ where z_1 is $\mathcal{P}$'s share and z_2 is $\mathcal{V}$'s share. According to the following analysis of main protocol Π_{AuthData}, we have that a *correct* pre-master secret pms is identical to $\mathit{AddModp}(\boldsymbol{z}_1 \oplus \boldsymbol{e}_0, \boldsymbol{z}_2)$ for an extra error $\boldsymbol{e}_0$ chosen by $\mathcal{A}$, where $\boldsymbol{z}_1$ (resp., $\boldsymbol{z}_2$) is the bit-decomposition of z_1 (resp., z_2). Let $e_0 \in \mathbb{Z}_p$ be an error defined by $\boldsymbol{e}_0$. Then we obtain that $z_1 + z_2 + e_0 = z + e_0 = \mathsf{pms} \in \mathbb{Z}_p$. Furthermore, we know that $z = \mathsf{F}_x(t_{\mathcal{P}} \cdot T_S + E + t_{\mathcal{V}} \cdot T_S) + e_z \in \mathbb{Z}_p$ for an adversarial-chosen error $E \in \mathbb{G}$. This means that $z = \mathsf{pms} + e' + e_z$ for some error e' associated with E and Z_2. Therefore, we have $e_z + e_0 + e' = 0$ except with negligible probability. This makes e_c, e_6, e_7, e_8, that are multiplied with a, have to be zero, since a is uniform in $\mathbb{Z}_p$ and has a sufficiently high entropy. Note that $Z_2 = (x_2, y_2) = t_{\mathcal{V}} \cdot T_S$ is computationally hard and has a sufficiently high entropy, which is implied by the PRF-ODH assumption. 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. Overall, we have $e_z = 0$, and thus $e_0 + e' = 0$. Since e' is computed from E and Z_2, we obtain that $E = 0$ and $e' = 0$, which makes $e_0 = 0$.

From $e_c, e_{c'}, e_r = 0$, we have $\langle \boldsymbol{g} * \boldsymbol{e}_1, \boldsymbol{b}_2 \rangle + \langle \boldsymbol{g} * \boldsymbol{e}_2, \boldsymbol{a}_2 \rangle = 0$, $\langle \boldsymbol{g} * \boldsymbol{e}_3, \boldsymbol{b}'_2 \rangle + \langle \boldsymbol{g} * \boldsymbol{e}_4, \boldsymbol{a}_2 \rangle = 0$ and $\langle \boldsymbol{g} * \boldsymbol{e}_5, \boldsymbol{r}_2 \rangle = 0$. 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 aborts. Therefore, $\mathcal{A}$ could reveal (on average) one-bit information 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 one-bit information on (x_2, y_2) or $z = \mathsf{pms}$. This is harmless as both of $Z_2 = (x_2, y_2)$ and pms have a sufficiently high entropy.

Simulation and analysis of sub-protocol Π_{PRF} (Figure 14). In the handshake phase (resp., the post-record phase), $\mathcal{S}_{\mathsf{PRF}}$ is given secret' (resp., secret^*) from the simulation of main protocol Π_{AuthData}. By emulating functionality $\mathcal{F}_{\mathsf{GP2PC}}$, $\mathcal{S}_{\mathsf{PRF}}$ interacts with adversary $\mathcal{A}$ as follows.

1. $\mathcal{S}_{\mathsf{PRF}}$ 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 $\mathcal{P}$'s 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 functionality $\mathcal{F}_{\mathsf{GP2PC}}$ 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 $\mathsf{type} =$ "open", $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by sending der' to $\mathcal{A}$. If $\mathsf{type} =$ "partial open", $\mathcal{S}_{\mathsf{PRF}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by parsing $\mathsf{der}' = (\mathsf{key}'_C, IV'_C, \mathsf{key}'_S, IV'_S)$ and sending (IV'_C, IV'_S) to $\mathcal{A}$.

5. In the post-record phase, $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by generating $IV_2^* = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus$ opad), and aborting if $IV_1' \neq f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{ipad})$.
6. For each $i \in [1, n]$, $\mathcal{S}_{\mathsf{PRF}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $M_i^* = f_{\mathsf{H}}(IV_2^*, W_i^*)$, where $W_i^* = f_{\mathsf{H}}(IV_1^*, M_{i-1}^*)$ and $M_0^* = label \| msg$, 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 $X_i^* = f_{\mathsf{H}}(IV_1^*, M_i^* \| label \| msg)$. If $\mathsf{type} =$ "open", then $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by aborting if $\mathsf{der}^* \neq \mathsf{der}'$. If $\mathsf{type} =$ "partial open", then $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by aborting if $IV_C^* \neq IV_C'$ or $IV_S^* \neq IV_S'$.

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 $M_0' = M_0^* = label \| msg$. If the protocol execution does not abort, then IV_1' and M_i' for each $i \in [1, n]$ are computed correctly, and thus $X_i = f_{\mathsf{H}}(IV_1', M_i' \| label \| msg) = X_i^*$ for each $i \in [1, n]$. Therefore, the simulation of checking der' if $\mathsf{type} =$ "open" and checking IV_C', IV_S' if $\mathsf{type} =$ "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 from the following analysis of main protocol Π_{AuthData}). In this case, we show that $\mathcal{A}$ cannot learn secret^* for all three type cases and der^* if $\mathsf{type} =$ "secret" and $(\mathsf{key}_C^*, \mathsf{key}_S^*)$ if $\mathsf{type} =$ "partial open" in the handshake phase. Based on the analysis in Section B.1, $\mathcal{A}$ leaks at most one-bit information of secret^* by sending "malicious" circuits to functionality $\mathcal{F}_{\mathsf{GP2PC}}$. In the ROM, $\mathcal{A}$ cannot learn secret^* from $IV_1' = IV_1^* = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{ipad})$ as secret^* has a sufficiently high entropy. Similarly, $\mathcal{A}$ cannot learn $IV_2^* = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{opad})$ 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 $\mathsf{type} =$ "open", or $(IV_C', IV_S') = (IV_C^*, IV_S^*)$ that is a part of der^* if $\mathsf{type} =$ "partial open". All the values do not reveal IV_2^* to $\mathcal{A}$ in the ROM. Therefore, if $\mathsf{type} =$ "secret", then $\mathsf{der}^* = \mathsf{PRF}_\ell(\mathsf{secret}^*, label, msg)$ is kept secret against $\mathcal{A}$, as secret^* and IV_2^* are unknown for $\mathcal{A}$, and $(label, msg)$ is different from that for other two type cases. If $\mathsf{type} =$ "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 $IV_1^* = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{ipad}) \in \{0,1\}^{256}$ 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} (Figures 15 and 16). We construct the sub-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 4.2. By emulating functionalities $\mathcal{F}_{\mathsf{GP2PC}}, \mathcal{F}_{\mathsf{OLEe}}, \mathcal{F}_{\mathsf{Com}}$, the sub-simulator $\mathcal{S}_{\mathsf{AEAD}}$ interacts with $\mathcal{A}$ as follows.

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 $\mathcal{A}$ if $\mathsf{type}_2 =$ "secret".

2. In the handshake/record phase, given an application key key' from the simulation of main protocol Π_{AuthData}, $\mathcal{S}_{\mathsf{AEAD}}$ simulates as follows:

   - If type_2 = "open", then $\mathcal{S}_{\mathsf{AEAD}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving a "malicious" circuit $f^{(1)}_{\mathsf{AES}}$, and then computing $(h'_{\mathcal{V}}, \mathrm{Z}'_{0,\mathcal{V}}, \mathrm{Z}'_1) \leftarrow f^{(1)}_{\mathsf{AES}}(\mathsf{key}')$ and sending Z'_1 to $\mathcal{A}$.
   - If type_2 = "secret", then $\mathcal{S}_{\mathsf{AEAD}}$ emulates $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving a "malicious" circuit $f^{(2)}_{\mathsf{AES}}$ 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 $(\mathrm{Z}'_{0,\mathcal{V}}, \ldots, \mathrm{Z}'_{n,\mathcal{V}}) \leftarrow f^{(2)}_{\mathsf{AES}}(\mathsf{key}', \mathrm{Z}'_{0,\mathcal{P}}, \ldots, \mathrm{Z}'_{n,\mathcal{P}}, \mathsf{st}'_0, \ldots, \mathsf{st}'_n)$.

3. In the handshake/record phase, $\mathcal{S}_{\mathsf{AEAD}}$ 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 $i \in [1, n]$, $\mathcal{S}_{\mathsf{AEAD}}$ 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, $\mathcal{S}_{\mathsf{AEAD}}$ sends $\mathrm{Z}'_{i,\mathcal{V}}$ to $\mathcal{A}$ and computes $\mathrm{C}_i := \mathrm{B}_i \oplus \mathrm{Z}'_{i,\mathcal{V}}$ for $i \in [1, n]$, and then sets $\mathbf{C} = (\mathrm{C}_1, \ldots, \mathrm{C}_n)$.
   - If type_1 = "decryption" and type_2 = "secret", then $\mathcal{S}_{\mathsf{AEAD}}$ sends $\mathrm{Z}'_{i,\mathcal{V}}$ for all $i \in [1, n]$ to $\mathcal{A}$.

4. If type_2 = "open", $\mathcal{S}_{\mathsf{AEAD}}$ simulates the generation of multiplication sharing $(\widetilde{h}_{\mathcal{P}}, \widetilde{h}_{\mathcal{V}})$ as follows:

   (a) $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{OLEe}}$ by receiving $\widetilde{h}'_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$, $s_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$ and an error vector $\boldsymbol{e}_1 \in (\mathbb{F}_{2^{128}})^{128}$ from $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ implicitly defines $(\widetilde{h}'_{\mathcal{P}})^{-1}$ as the $\mathcal{P}$'s input $\widetilde{h}_{\mathcal{P}}$ if $\widetilde{h}'_{\mathcal{P}} \neq 0$, or sets $\widetilde{h}_{\mathcal{P}} = 0$ otherwise.

   (b) $\mathcal{S}_{\mathsf{AEAD}}$ sets an error $e'_1 := \langle \boldsymbol{g} * \boldsymbol{e}_1, \boldsymbol{v}_1 \rangle \in \mathbb{F}_{2^{128}}$, and computes $\mathcal{V}$'s share $s_{\mathcal{V}}$ following the definition of $\mathcal{F}_{\mathsf{OLEe}}$, where $\boldsymbol{v}_1$ is the vector representation of $h'_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$.

   (c) $\mathcal{S}_{\mathsf{AEAD}}$ receives $d' \in \mathbb{F}_{2^{128}}$ from $\mathcal{A}$, and computes an error $e'_0 := d' - \widetilde{h}'_{\mathcal{P}} \cdot h_{\mathcal{P}} - s_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes the $\mathcal{V}$'s share $\widetilde{h}_{\mathcal{V}} := d' + s_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$.

5. If type_2 = "open", for each $i \in [2, m]$, $\mathcal{S}_{\mathsf{AEAD}}$ simulates the generation of $\left[h^i\right]_{2^{128}}$ as follows:

   (a) $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{OLEe}}$ by receiving $\widetilde{h}_{\mathcal{P},i} \in \mathbb{F}_{2^\lambda}$, $a_i \in \mathbb{F}_{2^\lambda}$ and an error vector $\boldsymbol{e}_i \in (\mathbb{F}_{2^\lambda})^\lambda$ from $\mathcal{A}$.

   (b) $\mathcal{S}_{\mathsf{AEAD}}$ defines an error $e'_i := \langle \boldsymbol{g} * \boldsymbol{e}_i, \boldsymbol{v}_i \rangle + (\widetilde{h}_{\mathcal{P},i} - (\widetilde{h}_{\mathcal{P}})^i) \cdot (\widetilde{h}_{\mathcal{V}})^i \in \mathbb{F}_{2^\lambda}$, where $\boldsymbol{v}_i$ is the vector representation of $(\widetilde{h}_{\mathcal{V}})^i \in \mathbb{F}_{2^{128}}$.

   (c) $\mathcal{S}_{\mathsf{AEAD}}$ computes the $\mathcal{V}$'s share on $\left[h^i\right]_{2^{128}}$ as $b_i = (\widetilde{h}_{\mathcal{P}} \cdot \widetilde{h}_{\mathcal{V}})^i - a_i + e'_i$.

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 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 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 an adversarial-chosen error $e_\sigma := \sigma'_{\mathcal{P}} - \sigma_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes an error $E := M(\widetilde{h}_{\mathcal{V}}, e'_0, e'_1) + L(e'_0, e'_1, \ldots, e'_m) + e_\sigma$ where M is a polynomial and L is a linear function depending on the protocol specification.

7. In the post-record phase, given key^* from the simulation of main protocol Π_{AuthData}, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (output) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as follows:
   - If 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 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^* from the simulation of Π_{AuthData}, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$ as follows:
   - If type_2 = "open", $\mathcal{S}_{\mathsf{AEAD}}$ 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 type_2 = "secret", $\mathcal{S}_{\mathsf{AEAD}}$ checks $\mathrm{z}'_0 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^*)$ 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 above check fails, $\mathcal{S}_{\mathsf{AEAD}}$ aborts.
9. If type_2 = "secret", for each $i \in [1, n]$, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the ($\mathsf{authinput}$) command of $\mathcal{F}_{\mathsf{IZK}}$ by receiving the MAC tag on $[\![\mathrm{z}_{i,\mathcal{P}}]\!]$, and then computes the MAC tag on $[\![\mathrm{M}_i]\!]$ following the protocol specification.

From the construction of $\mathcal{S}_{\mathsf{AEAD}}$, it is easy to see that the simulation of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ is perfect. It is also the case for the simulation of functionality $\mathcal{F}_{\mathsf{IZK}}$. During the emulation of functionality $\mathcal{F}_{\mathsf{OLEe}}$, $\mathcal{S}_{\mathsf{AEAD}}$ receives the values from $\mathcal{A}$ and extracts the errors introduced by $\mathcal{A}$. Thus, this emulation is also perfect. Below, suppose that the main protocol Π_{AuthData} does not abort and thus sub-protocol Π_{AEAD} does not abort as well. Otherwise, the ideal-world execution is natural to be indistinguishable from the real-world execution.

Since the circuit evaluation is verified correctly, we obtain that $h'_{\mathcal{V}} = \mathsf{AES}(\mathsf{key}^*, \mathbf{0}) \oplus h_{\mathcal{P}}$, $\mathrm{z}'_{0,\mathcal{V}} = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^*) \oplus \mathrm{z}_{0,\mathcal{P}}$ and $\mathrm{z}'_1 = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^* + 1)$ in the case of type_2 = "open". Here, key^* is a correct application key and has a sufficiently high entropy based on the following analysis of main protocol Π_{AuthData}. In particular, based on the analysis in Section B.1, $\mathcal{A}$ leaks at most one-bit information of key^* by sending a "malicious" circuit to functionality $\mathcal{F}_{\mathsf{GP2PC}}$. Furthermore, the AES blocks sent to $\mathcal{A}$ do not reveal key^* in the ICM. If type_2 = "secret", we also obtain that $\mathrm{z}'_{i,\mathcal{V}} = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^* + i) \oplus \mathrm{z}_{i,\mathcal{P}}$ for each $i \in [0, n]$. If $\mathcal{A}$ uses an incorrect circuit $f^{(1)}_{\mathsf{AES}}$ or $f^{(2)}_{\mathsf{AES}}$, or $\mathsf{key}' \neq \mathsf{key}^*$, then it has to guess the outputs of ideal cipher AES on the key^*, which happens with negligible probability in the Ideal Cipher Model (ICM). Note that $(h_{\mathcal{P}}, \mathrm{z}_{0,\mathcal{P}})$ or $(\mathrm{z}_{0,\mathcal{P}}, \mathrm{z}_{1,\mathcal{P}}, \ldots, \mathrm{z}_{n,\mathcal{P}})$ has been committed in the preprocessing phase. If the protocol does not abort, then the simulation w.r.t. the AEAD encryption/decryption is perfect. In particular, $\mathcal{S}_{\mathsf{AEAD}}$ succeeds to extract the plaintext query $\mathbf{M}$.

In the following, we analyze the simulation of generating GMAC tags. $\mathcal{S}_{\mathsf{AEAD}}$ always simulates the process honestly and extracts the errors introduced by $\mathcal{A}$. In the ICM, we have that $h'_{\mathcal{V}}$ is computationally indistinguishable from a uniform string in $\{0,1\}^{128}$ before checking the correctness of a GMAC tag. That is, $h = \mathsf{AES}(\mathsf{key}^*, \mathbf{0})$ is computationally indistinguishable from a uniform string. From the analysis of main protocol Π_{AuthData}, we know that the error E added into the final GMAC tag σ is equal to 0, as the TLS server or $\mathcal{V}$ checks the validity of σ. Note that $e'_i = \langle \boldsymbol{g} * \boldsymbol{e}_i, \boldsymbol{v}_i \rangle + t_i \cdot (\widetilde{h}_{\mathcal{V}})^i \in \mathbb{F}_{2^\lambda}$ for $i \in [1, m]$ and $E = M(\widetilde{h}_{\mathcal{V}}, e'_0, e'_1) + L(e'_0, e'_1, \ldots, e'_m) + e_\sigma = 0$, where $t_i = 0$ if $i = 1$ or $t_i = \widetilde{h}_{\mathcal{P},i} - (\widetilde{h}_{\mathcal{P}})^i$ if $i \neq 1$. If $\widetilde{h}'_{\mathcal{P}} \neq 0$, then $\widetilde{h}_{\mathcal{V}} = h \cdot \widetilde{h}'_{\mathcal{P}}$, which is computationally indistinguishable from a uniform string in $\{0,1\}^{128}$. In this case, we have that $t_i = 0$ for all $i \in [1, m]$, $e'_0 = 0$ and $e'_1 = 0$. Adversary $\mathcal{A}$ can obtain only one-bit information on $\widetilde{h}_{\mathcal{V}}$ from $e'_1 = 0$, $L(e'_0, e'_1, \ldots, e'_m) + e_\sigma = 0$ and $e'_i = \langle \boldsymbol{g} * \boldsymbol{e}_i, \boldsymbol{v}_i \rangle$ for $i \in [1, m]$, where an incorrect guess on a few bits of $\widetilde{h}_{\mathcal{V}}$ would incur that the protocol aborts. Equivalently, $\mathcal{A}$ leaks at most one-bit information of h.

If $\widetilde{h}'_{\mathcal{P}} = 0$, then $\widetilde{h}_{\mathcal{V}}$ is the value learned by $\mathcal{A}$, and thus $\mathcal{A}$ can obtain all the $\mathcal{V}$'s shares on $\left[h^i\right]_{2^{128}}$. For the case of $\mathsf{type}_1 =$ "encryption", $\mathcal{A}$ has to guess the uniform string $h \in \{0,1\}^{128}$ in order to pass the server's verification, which is negligible. For the case of $\mathsf{type}_1 =$ "decryption", the shares on a GMAC tag σ are first committed and then opened. In this case, before the shares are opened, $\mathrm{z}_{0,\mathcal{V}} = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^*) \oplus \mathrm{z}_{0,\mathcal{P}}$ is computationally indistinguishable from a uniform string in $\{0,1\}^{128}$ in the ICM. Adversary $\mathcal{A}$ has to guess the value $\mathrm{z}_{0,\mathcal{V}} \in \{0,1\}^{128}$ to pass the $\mathcal{V}$'s verification, which is also negligible. Overall, in both handshake and record phases, $\mathcal{A}$ cannot forge a GMAC tag on any message different from the original message.

Simulation and analysis of main protocol Π_{AuthData} (Figures 17 and 18). Simulator $\mathcal{S}$ plays the roles of both the honest verifier $\mathcal{V}$ and server. Given $\mathcal{S}_{\mathsf{E2F}}$, $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$, $\mathcal{S}$ emulates the functionalities used in protocol Π_{AuthData}, and interacts with 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}.
2. Following the TLS specification, on behalf of the server, $\mathcal{S}$ receives $\mathrm{REQ}_C = r_C$ from $\mathcal{A}$ and sends RES_S to $\mathcal{A}$, where (r_S, T_S) is included in RES_S. Then, on behalf of verifier $\mathcal{V}$, $\mathcal{S}$ receives $(\mathrm{REQ}_C \oplus \mathrm{E}_1, \mathrm{RES}_S \oplus \mathrm{E}_2)$ from $\mathcal{A}$ for two errors $\mathrm{E}_1, \mathrm{E}_2$ chosen by $\mathcal{A}$, and then checks that $\mathrm{E}_1 = 0$ and $\mathrm{E}_2 = 0$. [8] If the 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 $\mathrm{RES}_C{}' = T_C + \mathrm{E}_3 \in \mathbb{G}$ from $\mathcal{A}$ for an error E_3 chosen by $\mathcal{A}$.
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 $\mathsf{pms}_{\mathcal{V}} \oplus \mathrm{E}_4$ 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 a "malicious" circuit f_{ADD} from $\mathcal{A}$. Besides, $\mathcal{S}$ emulates the (commit) command of $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving $\mathsf{pms}_{\mathcal{P}} \oplus \mathrm{E}_5 \in \{0,1\}^{\lceil \log p \rceil}$ from $\mathcal{A}$. $\mathcal{S}$ stores $\mathsf{pms}_{\mathcal{P}} \oplus \mathrm{E}_5$ and computes $\mathsf{pms}' := f_{\mathsf{ADD}}(\mathsf{pms}_{\mathcal{V}} \oplus \mathrm{E}_4)$. Let $\mathsf{pms}' = \mathsf{pms} \oplus \mathrm{E}_6$. Here, $\mathrm{E}_4, \mathrm{E}_5, \mathrm{E}_6$ are three errors.
6. Given pms' and $r_C \| r_S$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(1)}_{\mathsf{PRF}}$ with $\mathsf{type} =$ "secret" and obtains a master secret $\mathsf{ms}' = \mathsf{ms} \oplus \mathrm{E}_7$ for an error E_7.
7. Given ms' and $r_S \| r_C$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(2)}_{\mathsf{PRF}}$ with $\mathsf{type} =$ "partial open". During this execution, $\mathcal{S}$ obtains application keys $(\mathsf{key}'_C, \mathsf{key}'_S) = (\mathsf{key}_C, \mathsf{key}_S) \oplus \mathrm{E}_8$ and sends $(IV'_C, IV'_S) = (IV_C, IV_S) \oplus \mathrm{E}_9$ to $\mathcal{A}$ for two errors $\mathrm{E}_8, \mathrm{E}_9$. Then, $\mathcal{S}$ initializes $(\mathsf{st}_C, \mathsf{st}_S) := (IV'_C, IV'_S)$.
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 $\mathcal{S}_{\mathsf{PRF}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(3)}_{\mathsf{PRF}}$ with $\mathsf{type} =$ "open". During this execution, $\mathcal{S}$ sends $\mathrm{UFIN}_C \oplus \mathrm{E}_{10}$ to $\mathcal{A}$ for an error E_{10}.
9. Given $(\mathsf{key}'_C, \mathsf{st}_C, \ell_C, \mathrm{H}_C, \mathrm{UFIN}_C \oplus \mathrm{E}_{10})$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(1)}_{\mathsf{AEAD}}$ with $\mathsf{type}_1 =$ "encryption" and $\mathsf{type}_2 =$ "open". During the sub-protocol execution, $\mathcal{S}$ (on behalf of $\mathcal{V}$) obtains $\mathrm{FIN}_C \oplus \mathrm{E}_{11} \oplus \delta_1$ for an error E_{11} and an adversarial-chosen error δ_1, and also makes $\mathcal{A}$ obtain $\mathrm{FIN}_C \oplus \mathrm{E}_{11}$. In addition, $\mathcal{S}$ gets the shares of both parties on $\left[h^i_C\right]_{2^{128}}$ for $i \in [1, m]$. On behalf of the server, $\mathcal{S}$ receives $(\mathrm{H}_C, \mathrm{FIN}_C \oplus \mathrm{E}_{12})$ from $\mathcal{A}$ for another error E_{12}, and then checks $\mathrm{FIN}_C \oplus \mathrm{E}_{12}$ following the TLS specification. If the check fails, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ updates $\mathsf{st}_C := \mathsf{st}_C + 2$.

[8] Without loss of generality, we assume that the error E_2 only changes the messages that have been signed, but does not modify the signatures. That is, we do not require strong unforgeability of the signature scheme.

10. $\mathcal{S}$ computes $\tau_S := \mathsf{H}(\text{REQ}_C \| \text{RES}_S \| \text{RES}_C \| \text{UFIN}_C \oplus \text{E}_{10})$. Given ms' and τ_S, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(4)}_{\mathsf{PRF}}$ with type = "open". During this execution, $\mathcal{S}$ sends $\text{UFIN}_S \oplus \text{E}_{13}$ to $\mathcal{A}$ for an error E_{13}.
11. On behalf of the server, $\mathcal{S}$ generates $(\text{H}_S, \text{FIN}_S)$ following the TLS specification and sends it to $\mathcal{A}$. On behalf of the verifier, $\mathcal{S}$ receives $(\text{H}_S, \text{FIN}_S \oplus \text{E}_{14})$ from $\mathcal{A}$ for an adversarial-chosen error E_{14}. Given $(\mathsf{key}'_S, \mathsf{st}_S, \ell_S, \text{H}_S, \text{FIN}_S \oplus \text{E}_{14})$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(2)}_{\mathsf{AEAD}}$ with type_1 = "decryption" and type_2 = "open". During this sub-protocol execution, $\mathcal{S}$ checks the correctness of $\text{FIN}_S \oplus \text{E}_{14}$ and sends $\text{UFIN}_S{}'$ to $\mathcal{A}$. Simulator $\mathcal{S}$ checks $\text{UFIN}_S \oplus \text{E}_{13} = \text{UFIN}_S{}'$. If any check fails, $\mathcal{S}$ aborts. $\mathcal{S}$ also gets the shares of both parties on $\left[h^i_S\right]_{2^{128}}$ for $i \in [1, m]$. Then, $\mathcal{S}$ updates $\mathsf{st}_S := \mathsf{st}_S + 2$.
12. Given $(\mathsf{key}'_C, \mathsf{st}_C, \ell_Q, \text{H}_Q)$ and the shares of both parties on $\left[h^i_C\right]_{2^{128}}$ for $i \in [1, m]$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the record phase of sub-protocol execution $\Pi^{(3)}_{\mathsf{AEAD}}$ with type_1 = "encryption" and type_2 = "secret". During this execution, $\mathcal{S}$ extracts a query Q and computes an AEAD ciphertext $\text{ENC}_Q \oplus \text{E}_{15}$ for an error E_{15}. 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 $(\text{H}_Q, \text{ENC}_Q \oplus \text{E}_{15} \oplus \delta_2)$ from $\mathcal{A}$ for an adversarial-chosen error δ_2, 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 $(\text{H}_R, \text{ENC}_R)$ with R following the TLS specification, and sends it to $\mathcal{A}$. On behalf of honest verifier $\mathcal{V}$, $\mathcal{S}$ receives $(\text{H}_R, \text{ENC}_R \oplus \text{E}_{16})$ from $\mathcal{A}$ for an adversarial-chosen error E_{16}.
14. $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the post-record phase of sub-protocol executions $\Pi^{(1)}_{\mathsf{AEAD}}$, $\Pi^{(2)}_{\mathsf{AEAD}}$ and $\Pi^{(3)}_{\mathsf{AEAD}}$. During this execution, $\mathcal{S}$ obtains $(h_C, h_S, \text{Z}_C, \text{Z}_S, \text{Z}_Q)$.
15. $\mathcal{S}$ emulates the ($\mathsf{revealandprove}$) command of $\mathcal{F}_{\mathsf{GP2PC}}$ by sending $\mathsf{pms}_{\mathcal{V}} \oplus \text{E}_4$ 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 $\mathsf{pms}_{\mathcal{P}} \oplus \text{E}_5$ stored previously and circuit $\overline{AddModp}$ to compute $\mathsf{pms}^* = AddModp(\mathsf{pms}_{\mathcal{P}} \oplus \text{E}_5, \mathsf{pms}_{\mathcal{V}} \oplus \text{E}_4)$.
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 $\Pi^{(1)}_{\mathsf{PRF}}$, $\Pi^{(2)}_{\mathsf{PRF}}$, $\Pi^{(3)}_{\mathsf{PRF}}$, $\Pi^{(4)}_{\mathsf{PRF}}$, $\Pi^{(1)}_{\mathsf{AEAD}}$, $\Pi^{(2)}_{\mathsf{AEAD}}$ and $\Pi^{(3)}_{\mathsf{AEAD}}$ to check the correctness of all values obtained by $\mathcal{V}$. If the check fails, then $\mathcal{S}$ aborts. During these executions, $\mathcal{S}$ computes ms^* and $(\mathsf{key}^*_C, \mathsf{st}^*_C, \mathsf{key}^*_S, \mathsf{st}^*_S)$.
18. $\mathcal{S}$ emulates the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by computing $\text{Z}_R := \mathsf{AES}(\mathsf{key}^*_S, \mathsf{st}^*_S)$ and receiving the MAC tags on $[\![\text{Z}_R]\!]$ from $\mathcal{A}$. After receiving Z'_R from $\mathcal{A}$, $\mathcal{S}$ emulates the (check) command of $\mathcal{F}_{\mathsf{IZK}}$ on $[\![\text{Z}_R]\!] - \text{Z}'_R$ by checking $\text{Z}'_R = \text{Z}_R$. Then, $\mathcal{S}$ checks the correctness of all GMAC tags in the AEAD ciphertexts $\text{FIN}_C \oplus \text{E}_{11} \oplus \delta_1, \text{FIN}_S \oplus \text{E}_{14}, \text{ENC}_Q \oplus \text{E}_{15}, \text{ENC}_R \oplus \text{E}_{16}$ following the protocol specification. If any check fails, $\mathcal{S}$ aborts.
19. $\mathcal{S}$ emulates the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving MAC tags on $[\![R]\!]$.
20. $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{Conv}}$ to convert $([\![Q]\!], [\![R]\!])$ into additively homomorphic commitments by sending commitment identifiers to $\mathcal{A}$.

The following analysis builds upon the analyses of sub-protocols Π_{E2F}, Π_{PRF} and Π_{AEAD}. From the above simulation, it is clear that all functionalities emulated by $\mathcal{S}$ behave just like as that in the real protocol execution. Based on the EUF-CMA security of signature scheme, we have that $\text{E}_1 = 0$ and $\text{E}_2 = 0$. Thus, the simulation of checking $\text{REQ}_C \oplus \text{E}_1$ and $\text{RES}_S \oplus \text{E}_2$ is computationally indistinguishable from that in the real protocol execution. From the simulation and analysis of sub-protocol Π_{E2F}, we have that $\mathcal{A}$ can only add an error into $\mathcal{V}$'s share $\mathsf{pms}_{\mathcal{V}}$, but cannot change $\mathcal{V}$'s

share to a known value in the handshake phase. The (prove) command of $\mathcal{F}_{\mathsf{GP2PC}}$ guarantees that all values obtained by $\mathcal{V}$ are computed correctly, and the valid and high-entropy share $\mathsf{pms}_{\mathcal{V}}$ is involved in the computation of pms^* and subsequent computation. If $\mathrm{E}_3 \neq 0$, then $\mathcal{A}$ has to guess the value $\mathrm{FIN}_S{}' = \mathsf{Func}_1(\mathit{AddModp}(\mathsf{pms}_{\mathcal{P}} \oplus \mathrm{E}_5, \mathsf{pms}_{\mathcal{V}} \oplus \mathrm{E}_4))$ before $\mathcal{V}$ receives the server finished message, where Func_1 is a function to compute the server finished message from scratch based on PRF and AES. Note that $\mathsf{pms}_{\mathcal{V}} \in \{0,1\}^{256}$ has the same entropy as $\mathsf{pms} = \mathsf{F}_x(t_{\mathcal{V}} \cdot T_S + t_{\mathcal{P}} \cdot T_S)$. According to the analysis of sub-protocol Π_{E2F} along with the analysis in Section B.1, $\mathsf{pms} \in \{0,1\}^{256}$ leaks one-bit information in the handshake and record phases, but has still a sufficiently high entropy. In this case, $\mathrm{FIN}_S{}'$ is unpredictable under the PRF-ODH assumption in the ROM. Therefore, the probability that $\mathrm{E}_3 \neq 0$ is negligible.

Since $\mathrm{E}_1, \mathrm{E}_2, \mathrm{E}_3$ are 0 with overwhelming probability based on the above analysis, we have that the client finished message is always computed correctly by the server. In the following analysis, we always assume that the protocol execution does not abort. Otherwise, the simulation is natural to be indistinguishable from the real protocol execution. Therefore, $\mathrm{FIN}_C \oplus \mathrm{E}_{12}$ received by the server is identical to the client finished message computed with the valid pre-master secret pms. This means that $\mathrm{E}_{12} = 0$. Let $\mathrm{FIN}_C \oplus \mathrm{E}_{11} = \mathsf{Func}_2(\mathsf{pms} \oplus \mathrm{E}_6)$ where Func_2 is a function to compute the client finished message using PRF and AES. Furthermore, according to the server's computation, we have $\mathrm{FIN}_C = \mathsf{Func}_2(\mathsf{pms})$, meaning that $\mathsf{Func}_2(\mathsf{pms} \oplus \mathrm{E}_6) \oplus \mathrm{E}_{11} = \mathsf{Func}_2(\mathsf{pms})$. Note that $\mathcal{A}$ knows the value FIN_C as it gets $\mathrm{FIN}_C \oplus \mathrm{E}_{12}$ and $\mathrm{E}_{12} = 0$, and thus it knows E_{11} as it also obtains $\mathrm{FIN}_C \oplus \mathrm{E}_{11}$. Based on the analysis of sub-protocols Π_{E2F} and Π_{PRF}, we obtain that pms reveals only one-bit information and still have a sufficiently high entropy in the handshake phase. In the ROM and ICM, we obtain $\mathrm{E}_6 = 0$ and $\mathrm{E}_{11} = 0$, except with negligible probability. Thus, the equality $\mathsf{pms} = \mathit{AddModp}(\mathsf{pms}_{\mathcal{P}} \oplus \mathrm{E}_5, \mathsf{pms}_{\mathcal{V}} \oplus \mathrm{E}_4)$ holds. According to the analysis of sub-protocol Π_{E2F}, we obtain that both E_4 and E_5 are identical to zero. Based on the fact that all values obtained by $\mathcal{V}$ are proven via calling functionality $\mathcal{F}_{\mathsf{GP2PC}}$, we further have that all of $\mathrm{E}_7, \mathrm{E}_8, \mathrm{E}_9, \mathrm{E}_{10}, \mathrm{E}_{13}$ are equal to zero. Furthermore, we also have that the AES blocks $h_C, h_S, \mathrm{Z}_C, \mathrm{Z}_S, \mathrm{Z}_Q, \mathrm{Z}_R$ obtained by $\mathcal{V}$ are correct, where these blocks are proved by calling functionality $\mathcal{F}_{\mathsf{GP2PC}}$. The GMAC tags in all AEAD ciphertexts obtained by $\mathcal{V}$ are verified using these AES blocks. Hence, we have that $\delta = 0$. Besides, the value $\mathrm{FIN}_S \oplus \mathrm{E}_{14}$ received by $\mathcal{V}$ is identical to FIN_S, meaning that $\mathrm{E}_{14} = 0$.

Note that all AES blocks, related to $\mathrm{ENC}_Q \oplus \mathrm{E}_{15}$ and $\mathrm{ENC}_R \oplus \mathrm{E}_{16}$, are valid based on the above analysis. According to the analysis of sub-protocol Π_{AEAD}, the query Q is extracted successfully and in turn sent to functionality $\mathcal{F}_{\mathsf{AuthData}}$. Together with that $\mathcal{V}$ checks the correctness of GMAC tag in ENC_Q, we have $\mathrm{E}_{15} = 0$. Following the analysis of sub-protocol Π_{AEAD}, we know that both of key_C, h_C leak only one-bit information in the handshake and record phases. In the ICM, it is infeasible that $\mathcal{A}$ forges a GMAC tag in the record phase, which means that $\delta_2 = 0$. Furthermore, we also have that key_S and h_S leak at most one-bit information, and the corresponding GMAC tags are unforgeable, in the handshake and record phases. Therefore, $\mathcal{A}$ cannot tamper the response R encrypted in ENC_R, where the correctness of ENC_R is verified by $\mathcal{V}$. In other words, we have $\mathrm{E}_{15} = 0$. The IT-MACs on the query Q and response R are obtained by calling the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$. Thus, these IT-MACs commit to the consistent Q and R. Furthermore, by calling functionality $\mathcal{F}_{\mathsf{Conv}}$, the values Q and R involved in additively homomorphic commitments are consistent. Overall, the output of honest $\mathcal{V}$ and adversary $\mathcal{A}$ in the real-world execution is computationally indistinguishable from that of honest $\mathcal{V}$ and simulator $\mathcal{S}$ in the ideal-world execution.

E.2 Proof for Malicious Verifier

We first show the constructions of sub-simulators $\mathcal{S}_{\mathsf{E2F}}$, $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$ for three sub-protocols, and then give the construction of simulator $\mathcal{S}$ 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 $\mathcal{S}$ is given access to functionality $\mathcal{F}_{\mathsf{AuthData}}$. Similarly, to analyze the security of each sub-protocol, we always cast it to the main protocol Π_{AuthData}.

Simulation and analysis of sub-protocol Π_{E2F} (Figure 13). $\mathcal{S}_{\mathsf{E2F}}$ 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 an input tuple $(b_2, a_2, b_2', a_2, r_2)$ and $\mathcal{V}$'s shares on $[a_1b_2]_p, [a_2b_1]_p$, $[a_1b_2']_p, [a_2b_1']_p$ and $[r_1r_2]_p$.
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 elements in $\mathbb{Z}_p$ to simulate the Open procedure on the values $\epsilon_1, \epsilon_2, \epsilon_3, w$, and receives the values sent by $\mathcal{A}$ during the Open procedure. Following the protocol specification, $\mathcal{S}_{\mathsf{E2F}}$ computes all $\mathcal{V}$'s shares in the handshake phase.
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 simulation shown in Section E.1.
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.

In the following analysis, suppose that the main-protocol execution does not abort and thus sub-protocol Π_{E2F} does not abort. Let $[z]_p = (z_1, z_2)$ such that $z_1 + z_2 = z \mod p$, and $\boldsymbol{z}_1$ (resp., $\boldsymbol{z}_2$) be the bit-decomposition of z_1 (resp., z_2). According to the following analysis of main protocol Π_{AuthData}, we have that $\mathsf{pms} = \mathit{AddModp}(\boldsymbol{z}_1, \boldsymbol{z}_2 \oplus \boldsymbol{e}_0)$ for an extra error $\boldsymbol{e}_0$ chosen by $\mathcal{A}$. Then we obtain that $z_1 + z_2 + e_0 = z + e_0 = \mathsf{pms} \in \mathbb{Z}_p$ where $e_0 \in \mathbb{Z}_p$ is an error defined by $\boldsymbol{e}_0$. Furthermore, we know that $z = \mathsf{F}_x(t_{\mathcal{P}} \cdot T_S + t_{\mathcal{V}} \cdot T_S + E) + e_z \in \mathbb{Z}_p$ for an adversarial-chosen error $E \in \mathbb{G}$. This means that $z = \mathsf{pms} + e' + e_z$ for some error e' associated with E and Z_1. Therefore, we have $e_z + e_0 + e' = 0$ except with negligible probability. This makes e_6, e_7, e_8, which are multiplied with a, have to be zero, since a is uniform in $\mathbb{Z}_p$. Note that $Z_1 = (x_1, y_1) = t_{\mathcal{P}} \cdot T_S$ is computationally hard and has a sufficiently high entropy under the PRF-ODH assumption. 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_9 multiplied with r has to be zero. Overall, we have $e_z = 0$, and thus $e_0 + e' = 0$. Since e' is computed from E and Z_1, we obtain that $E = 0$ and $e' = 0$, which makes $e_0 = 0$. In the real protocol execution, $\mathcal{P}$'s shares a_1, b_1, b_1', r_1 are uniformly random in $\mathbb{Z}_p$, and thus the values sent by $\mathcal{P}$ during the Open procedure are random. Thus, the simulation for the Open procedure is perfect. It means that sub-protocol Π_{E2F} does not reveal any information on Z_1.

Simulation and analysis of sub-protocol Π_{PRF} (Figure 14). $\mathcal{S}_{\mathsf{PRF}}$ is given a possible incorrect secret' (resp., a correct secret^*) from the simulation of main protocol Π_{AuthData} in the handshake phase (resp., the post-record phase). $\mathcal{S}_{\mathsf{PRF}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$, and interacts with $\mathcal{A}$.

1. $\mathcal{S}_{\mathsf{PRF}}$ emulates the (eval) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $IV_2 = f_{\mathsf{H}}(IV_0, \mathsf{secret}' \oplus \mathrm{opad})$, receiving an error e_1 from $\mathcal{A}$ and sending $IV_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret}' \oplus \mathrm{ipad})$ to $\mathcal{A}$. Then $\mathcal{S}_{\mathsf{PRF}}$ sets $IV_1' := IV_1 \oplus e_1$ that is implicitly opened to honest $\mathcal{P}$.
2. From $i = 1$ to n, $\mathcal{S}_{\mathsf{PRF}}$ emulates the (eval) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving an error $e_{2,i}$ from $\mathcal{A}$, 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 $\mathcal{P}$.

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 the (eval) command of 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 type = "open", $\mathcal{S}_{\mathsf{PRF}}$ emulates the (output) command of 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 type = "partial open", $\mathcal{S}_{\mathsf{PRF}}$ emulates the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ 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, in the simulation of main protocol Π_{AuthData}, the simulator $\mathcal{S}$ computes $IV_2^* = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{opad})$, and checks the following values that are computed correctly with secret^* and IV_2^*.

$$
\begin{aligned}
&IV'_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{ipad});\\
&M'_i = f_{\mathsf{H}}(IV_2^*, W'_i) \text{ where } W'_i = f_{\mathsf{H}}(IV'_1, M'_{i-1});\\
&\text{If } \mathsf{type}\text{="open"}, \mathsf{der}' = \big(f_{\mathsf{H}}(IV_2^*, X'_1), \ldots, f_{\mathsf{H}}(IV_2^*, X'_{n-1}),\\
&\mathsf{Trunc}_m(f_{\mathsf{H}}(IV_2^*, X'_n))\big) \text{ where } X'_i = f_{\mathsf{H}}(IV'_1, M'_i \| \mathit{label} \| \mathit{msg});\\
&\text{If } \mathsf{type}\text{="partial open"}, IV'_C = IV_C^* \text{ and } IV'_S = IV_S^*, \text{ where}\\
&IV_C^*, IV_S^* \text{ are computed with } IV_2^*, X'_1, \ldots, X'_n.
\end{aligned}
$$

   If any check fails, $\mathcal{S}$ 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 $\mathcal{A}$.

Clearly, the simulation for the (eval) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ is perfect. If the execution of main protocol Π_{AuthData} does not abort, then $\mathsf{secret}' = \mathsf{secret}^*$ and thus $IV_2 = IV_2^*$, based on the following analysis of Π_{AuthData}. 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. Therefore, due to the local check performed by $\mathcal{P}$, all of the errors $e_1, e_{2,1}, \ldots, e_{2,n}, e_3, e_4, e_5$ are equal to zero, where the analysis is easy to be done by induction. 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.

Below, we show that $\mathcal{A}$ cannot learn any information on secret^* for all three type cases and der^* if type = "secret" and $(\mathsf{key}_C^*, \mathsf{key}_S^*)$ if type = "partial open". In the ROM, $\mathcal{A}$ cannot learn secret^* from $IV_1 = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{ipad})$ as secret^* has a sufficiently high entropy. Similarly, $\mathcal{A}$ cannot learn $IV_2^* = f_{\mathsf{H}}(IV_0, \mathsf{secret}^* \oplus \mathrm{opad})$ from IV_1 in the ROM, where $\mathrm{ipad} \neq \mathrm{opad}$. That is, $IV_2^* \in \{0,1\}^{256}$ is uniform. In the handshake phase, $\mathcal{A}$ obtains $M_i = f_{\mathsf{H}}(IV_2^*, W'_i)$ for $i \in [1, n]$. In addition, $\mathcal{A}$ gets $\mathsf{der} = \big(f_{\mathsf{H}}(IV_2^*, X'_1), \ldots, f_{\mathsf{H}}(IV_2^*, X'_{n-1}), \mathsf{Trunc}_m(f_{\mathsf{H}}(IV_2^*, X'_n))\big)$ if type = "open", or (IV_C^*, IV_S^*) that is a part of der if type = "partial open". All the values do not reveal IV_2^* to $\mathcal{A}$ in the ROM. 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 type cases. 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, these secret values are kept unknown against $\mathcal{A}$.

Simulation and analysis of sub-protocol Π_{AEAD} (Figures 15 and 16). Similarly, we construct the sub-simulator $\mathcal{S}_{\mathsf{AEAD}}$ for all four cases related to type_1 and type_2. Specifically, $\mathcal{S}_{\mathsf{AEAD}}$ 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 handshake/record phase, given an application key key' and an AEAD state st' from the simulation of main protocol Π_{AuthData}, $\mathcal{S}_{\mathsf{AEAD}}$ simulates as follows:
   - If type_2 = "open", then $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (eval) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $(h_{\mathcal{V}}, \mathrm{Z}_{0,\mathcal{V}}, \mathrm{Z}_1) \leftarrow \mathcal{C}_{aes}(\mathsf{key}', h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}}, \mathsf{st}', \mathsf{st}'+1)$, where $h_{\mathcal{P}}, \mathrm{Z}_{0,\mathcal{P}} \in \mathbb{F}_{2^{128}}$ are sampled uniformly by $\mathcal{S}_{\mathsf{AEAD}}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ by sending $(h_{\mathcal{V}}, \mathrm{Z}_{0,\mathcal{V}}, \mathrm{Z}_1)$ to $\mathcal{A}$, and 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 the (eval) command of $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $(\mathrm{Z}_{0,\mathcal{V}}, \ldots, \mathrm{Z}_{n,\mathcal{V}}) \leftarrow \mathcal{D}_{aes}(\mathsf{key}', \mathrm{Z}_{0,\mathcal{P}}, \ldots, \mathrm{Z}_{n,\mathcal{P}}, \mathsf{st}', \mathsf{st}'+1, \ldots, \mathsf{st}'+n)$ where $\mathrm{Z}_{0,\mathcal{P}}, \ldots, \mathrm{Z}_{n,\mathcal{P}} \in \mathbb{F}_{2^{128}}$ are sampled at random by $\mathcal{S}_{\mathsf{AEAD}}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (output) command of $\mathcal{F}_{\mathsf{GP2PC}}$ by sending $\mathrm{Z}_{i,\mathcal{V}}$ for $i \in [0, n]$ to $\mathcal{A}$.
2. 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", for each $i \in [1, n]$, $\mathcal{S}_{\mathsf{AEAD}}$ samples $\mathrm{B}_i \leftarrow \mathbb{F}_{2^{128}}$ and sends B_i to $\mathcal{A}$. Then, for each $i \in [1, n]$, $\mathcal{S}_{\mathsf{AEAD}}$ receives $\mathrm{Z}'_{i,\mathcal{V}} = \mathrm{Z}_{i,\mathcal{V}} \oplus e_{z,i}$ from $\mathcal{A}$, where $e_{z,i}$ is an adversarial-chosen error, and then computes the ciphertext $\mathbf{C}'$ following the protocol specification.
   - If type_1 = "decryption" and type_2 = "secret", then given $\mathbf{C} = (\mathrm{C}_1, \ldots, \mathrm{C}_n)$, $\mathcal{S}_{\mathsf{AEAD}}$ receives $\mathrm{Z}'_{i,\mathcal{V}} = \mathrm{Z}_{i,\mathcal{V}} \oplus e_{z,i}$ for all $i \in [1, n]$ from $\mathcal{A}$.
3. If type_2 = "open", $\mathcal{S}_{\mathsf{AEAD}}$ simulates the computation of multiplication sharing $(\widetilde{h}_{\mathcal{P}}, \widetilde{h}_{\mathcal{V}})$ as follows:
   (a) $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{OLEe}}$ by receiving $h'_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$ and $s_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$ from $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ sets an error $e_1 := h'_{\mathcal{V}} - h_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$, where $\widetilde{h}_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$ is sampled at random by $\mathcal{S}_{\mathsf{AEAD}}$. Next, $\mathcal{S}_{\mathsf{AEAD}}$ computes the $\mathcal{P}$'s share $s_{\mathcal{P}} := (\widetilde{h}_{\mathcal{P}})^{-1} \cdot h_{\mathcal{V}} - s_{\mathcal{V}} + e_1 \cdot (\widetilde{h}_{\mathcal{P}})^{-1} \in \mathbb{F}_{2^{128}}$.
   (b) $\mathcal{S}_{\mathsf{AEAD}}$ samples $d \leftarrow \mathbb{F}_{2^{128}}$, and sends $d + e_1 \cdot (\widetilde{h}_{\mathcal{P}})^{-1} \in \mathbb{F}_{2^{128}}$ to $\mathcal{A}$, and then computes the $\mathcal{V}$'s share $\widetilde{h}_{\mathcal{V}}$ following the protocol specification.
4. If type_2 = "open", for each $i \in [2, m]$, $\mathcal{S}_{\mathsf{AEAD}}$ simulates the computation of $\left[h^i\right]_{2^{128}}$ as follows:
   (a) $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{OLEe}}$ by receiving $\widetilde{h}_{\mathcal{V},i} \in \{0,1\}^{128}$ and $b_i \in \{0,1\}^{128}$ from $\mathcal{A}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ defines an error $e_i := \widetilde{h}_{\mathcal{V},i} - (\widetilde{h}_{\mathcal{V}})^i \in \mathbb{F}_{2^{128}}$.
   (b) $\mathcal{S}_{\mathsf{AEAD}}$ computes the $\mathcal{P}$'s share on $\left[h^i\right]_{2^{128}}$ as $a_i := (\widetilde{h}_{\mathcal{P}} \cdot \widetilde{h}_{\mathcal{V}})^i - b_i + e_i \cdot (\widetilde{h}_{\mathcal{P}})^i \in \mathbb{F}_{2^{128}}$.
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 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 $\mathrm{CT} = (\mathbf{C}, \sigma)$.
   - If 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 CT given to $\mathcal{S}_{\mathsf{AEAD}}$. If the check fails, $\mathcal{S}_{\mathsf{AEAD}}$ aborts.

   In both cases, $\mathcal{S}_{\mathsf{AEAD}}$ computes an adversarial-chosen error $e_\sigma := \sigma'_{\mathcal{V}} - \sigma_{\mathcal{V}} \in \mathbb{F}_{2^{128}}$. Then, $\mathcal{S}_{\mathsf{AEAD}}$ computes an error $E := \sum_{i \in [2,m]} p_i(e_1, e_i) \cdot (\widetilde{h}_{\mathcal{P}})^i + e_\sigma$, where p_i is a polynomial depending on the protocol specification. The error E is added into the resulting GMAC tag.

6. In the post-record phase, given key^* from the simulation of main protocol Π_{AuthData}, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (output) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ as follows:
   - If 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 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. After obtaining $(\mathsf{key}^*, \mathsf{st}^*)$, the simulator $\mathcal{S}$ towards main protocol Π_{AuthData} performs the local verification to check the correctness of all AEAD ciphertexts. If the check fails, $\mathcal{S}$ aborts. This step would be done in the simulation of main protocol Π_{AuthData}.
8. If type_2 = "secret", $\mathcal{S}_{\mathsf{AEAD}}$
9. If type_2 = "secret", for each $i \in [1, n]$, $\mathcal{S}_{\mathsf{AEAD}}$ emulates the (authinput) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving the key on $[\![\mathrm{z}_{i,\mathcal{P}}]\!]$, and then computes the key on $[\![\mathrm{M}_i]\!]$ following the protocol specification.

For the emulation of functionality $\mathcal{F}_{\mathsf{OLEe}}$, $\mathcal{S}_{\mathsf{AEAD}}$ simply receives the values from $\mathcal{A}$ and extracts the errors chosen by $\mathcal{A}$, which means that the emulation is perfect. From the construction of $\mathcal{S}_{\mathsf{AEAD}}$, it is clear that functionality $\mathcal{F}_{\mathsf{GP2PC}}$ is emulated perfectly, as $\mathcal{S}_{\mathsf{AEAD}}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ following the definition of $\mathcal{F}_{\mathsf{GP2PC}}$ and using the actual values. It is straightforward to see that the simulation of functionality $\mathcal{F}_{\mathsf{IZK}}$ is perfect. As such, suppose that the main protocol Π_{AuthData} does not abort and thus sub-protocol Π_{AEAD} does not abort as well.

According to the local verification (step 14) in main protocol Π_{AuthData}, we have that $\mathsf{key}' = \mathsf{key}^*$ and $\mathsf{st}' = \mathsf{st}^*$ except with negligible probability, since $\mathsf{key}^* \in \{0,1\}^{128}$ is computationally indistinguishable from a uniform string and AES is an ideal cipher. Therefore, with overwhelming probability, $e_z = 0$ if type_2 = "open" and $e_{z,i} = 0$ for all $i \in [1, n]$ if type_2 = "secret". Furthermore, we have that $h, \mathrm{z}_0, \mathrm{z}_1$ computed by $\mathcal{S}_{\mathsf{AEAD}}$ are valid if type_2 = "open", and $\mathrm{z}_{i,\mathcal{V}}$ for all $i \in [0, n]$ sent to $\mathcal{A}$ are computed correctly with key^* and st^* if type_2 = "secret". It is easy to see that the AEAD encryption/decryption simulated by $\mathcal{S}_{\mathsf{AEAD}}$ is perfect, except that $\mathrm{B}_i \in \{0,1\}^{128}$ for each $i \in [1, n]$ is sampled uniformly for type_1 = "encryption" and type_2 = "secret". In the real protocol execution, for each $i \in [1, n]$, B_i is masked by $\mathrm{z}_{i,\mathcal{P}}$. For each $i \in [1, n]$, we have that $\mathrm{z}_{i,\mathcal{P}} = \mathsf{AES}(\mathsf{key}^*, \mathsf{st}^* + i) \oplus \mathrm{z}_{i,\mathcal{V}}$, which is computationally indistinguishable from a uniform string, since key^* is uniform and AES is an ideal cipher. Therefore, B_i for all $i \in [1, n]$ simulated by $\mathcal{S}$ are computationally indistinguishable from that in the real protocol execution.

Below, we analyze the simulation of GMAC tags. $\mathcal{S}_{\mathsf{AEAD}}$ always simulates the process honestly and extracts the errors chosen by $\mathcal{A}$. In the ICM, we have that $h_{\mathcal{P}} = \mathsf{AES}(\mathsf{key}^*, \mathbf{0}) \oplus h_{\mathcal{V}}$ is computationally indistinguishable from a uniform string in $\{0,1\}^{128}$. Together with that $\widetilde{h}_{\mathcal{P}} \in \mathbb{F}_{2^{128}}$ is sampled at random, we have that $d = (\widetilde{h}_{\mathcal{P}})^{-1} \cdot h_{\mathcal{P}} + s_{\mathcal{P}}$ is computationally indistinguishable from a uniform element in $\mathbb{F}_{2^{128}}$, even though $\mathcal{A}$ learns $s_{\mathcal{P}}$ by setting $h'_{\mathcal{V}} = 0$. Thus, the simulation of d is computationally indistinguishable from the real protocol execution. Since the protocol execution does not abort, we obtain that the error $E = \sum_{i \in [2,m]} p_i(e_1, e_i) \cdot (\widetilde{h}_{\mathcal{P}})^i + e_\sigma = 0$, where e_1, e_i are the errors chosen by $\mathcal{A}$. Even if $\mathcal{A}$ obtains $d + e_1$, we still have that $\widetilde{h}_{\mathcal{P}}$ is computationally indistinguishable from a uniform element in $\mathbb{F}_{2^{128}}$, due to the uniformity of $h_{\mathcal{P}}$. Therefore, we have that $p_i(e_1, e_i) = 0$ for all $i \in [2, m]$ and $e_\sigma = 0$. Based on the fact that both h and z_0 are computationally indistinguishable from uniform strings in $\{0,1\}^{128}$, it is infeasible that $\mathcal{A}$ forges a GMAC tag.

Simulation and analysis of main protocol Π_{AuthData} (Figures 17 and 18). Simulator $\mathcal{S}$ plays the roles of both the honest prover $\mathcal{P}$ and server. By invoking $\mathcal{S}_{\mathsf{E2F}}$, $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$ constructed as above, $\mathcal{S}$ emulates the functionalities used in protocol Π_{AuthData}, and interacts with $\mathcal{A}$ as follows.

1. $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{E2F}}$ (resp., $\mathcal{S}_{\mathsf{AEAD}}$) to simulate the preprocessing phase of sub-protocol Π_{E2F} (resp., Π_{AEAD}). $\mathcal{S}$ also emulates functionality $\mathcal{F}_{\mathsf{ZK}}$ by receiving a global key $\Delta \in \{0,1\}^{\lambda}$ from $\mathcal{A}$.
2. Following the protocol specification, $\mathcal{S}$ (on behalf of $\mathcal{P}$) sends REQ_C to $\mathcal{A}$. Then, on behalf of the server, $\mathcal{S}$ receives $\text{REQ}_C \oplus \text{E}_1$ from $\mathcal{A}$ where E_1 is an adversarial-chosen error. Following the protocol specification, $\mathcal{S}$ (on behalf of the server) computes RES_S according to $\text{REQ}_C \oplus \text{E}_1$ and sends it to $\mathcal{A}$, where $\mathcal{S}$ records $t_S \in \mathbb{Z}_q$ for $T_S = t_S \cdot G$ contained in RES_S. Then, on behalf of $\mathcal{P}$, $\mathcal{S}$ receives $\text{RES}_S \oplus \text{E}_2$ from $\mathcal{A}$, and forwards it to the verifier controlled by $\mathcal{A}$, where E_2 is an error chosen by $\mathcal{A}$. Then $\mathcal{S}$ checks that $\text{E}_1 = 0$ and $\text{E}_2 = 0$ and aborts if the check fails.
3. After receiving $T_{\mathcal{V}} \in \mathbb{G}$, $\mathcal{S}$ samples $t_{\mathcal{P}} \leftarrow \mathbb{Z}_q$, computes $T_{\mathcal{P}} := t_{\mathcal{P}} \cdot G$ and sends $\text{RES}_C = T_C = T_{\mathcal{P}} + T_{\mathcal{V}}$ to $\mathcal{A}$. On behalf of the server, $\mathcal{S}$ receives $\text{RES}_C + \text{E}_3 \in \mathbb{G}$ from $\mathcal{A}$ for an error E_3 chosen by $\mathcal{A}$.
4. $\mathcal{S}$ 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, $\mathcal{S}$ computes $\widetilde{\mathsf{pms}}'_{\mathcal{P}} \in \mathbb{Z}_p$ following the specification of sub-protocol Π_{E2F}.
5. $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving $\mathsf{pms}_{\mathcal{V}} \oplus \text{E}_4 \in \{0,1\}^{\lceil \log p \rceil}$ from $\mathcal{A}$. Then, $\mathcal{S}$ defines $\mathsf{pms}_{\mathcal{P}} \oplus \text{E}_5$ as the bit-decomposition of $\widetilde{\mathsf{pms}}'_{\mathcal{P}} \in \mathbb{Z}_p$, and computes $\mathsf{pms}' := \mathit{AddModp}(\mathsf{pms}_{\mathcal{P}} \oplus \text{E}_5, \mathsf{pms}_{\mathcal{V}} \oplus \text{E}_4)$. Let $\mathsf{pms}' = \mathsf{pms} \oplus \text{E}_6$ for an error E_6.
6. Given pms' and $r_C \| r_S$ where r_C is included in REQ_C and r_S is involved in $\text{RES}_S \oplus \text{E}_2$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(1)}_{\mathsf{PRF}}$ with type = "secret" and obtains a master secret $\mathsf{ms}' = \mathsf{ms} \oplus \text{E}_7$ for an error E_7.
7. Given ms' and $r_S \| r_C$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(2)}_{\mathsf{PRF}}$ with type = "partial open". During this execution, $\mathcal{S}$ obtains $(\mathsf{key}'_C, \mathsf{key}'_S) = (\mathsf{key}_C, \mathsf{key}_S) \oplus \text{E}_8$, and receives $(IV'_C, IV'_S) = (IV_C, IV_S) \oplus \text{E}_9$ from $\mathcal{A}$ where E_8, E_9 are two errors. Then, $\mathcal{S}$ initializes $(\mathsf{st}_C, \mathsf{st}_S) = (IV_C{}', IV_S{}')$.
8. $\mathcal{S}$ computes $\tau_C := \mathsf{H}(\text{REQ}_C \| \text{RES}_S \| \text{RES}_C)$. Given ms' and τ_C, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(3)}_{\mathsf{PRF}}$ with type = "open". During this execution, by emulating functionality $\mathcal{F}_{\mathsf{GP2PC}}$, $\mathcal{S}$ receives $\text{UFIN}_C \oplus \text{E}_{10}$ from $\mathcal{A}$ for an error E_{10}.
9. Given $(\mathsf{key}'_C, \mathsf{st}_C, \ell_C, \text{H}_C, \text{UFIN}_C \oplus \text{E}_{10})$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(1)}_{\mathsf{AEAD}}$ with type_1 = "encryption" and type_2 = "open". During this execution, $\mathcal{S}$ obtains $\text{FIN}_C \oplus \text{E}_{11}$ for an error E_{11} and the shares of both parties on $\left[h^i_C\right]_{2^{128}}$ for $i \in [1, m]$. On behalf of the server, $\mathcal{S}$ receives $(\text{H}_C, \text{FIN}_C \oplus \text{E}_{12})$ from $\mathcal{A}$ for another error E_{12}, and checks its correctness following the TLS specification. If the check fails, $\mathcal{S}$ aborts. Then, $\mathcal{S}$ updates $\mathsf{st}_C := \mathsf{st}_C + 2$.
10. $\mathcal{S}$ computes $\tau_S := \mathsf{H}(\text{REQ}_C \| \text{RES}_S \| \text{RES}_C \| \text{UFIN}_C \oplus \text{E}_{10})$. Given ms' and τ_S, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ to simulate the handshake phase of sub-protocol execution $\Pi^{(4)}_{\mathsf{PRF}}$ with type = "open". During this execution, by emulating functionality $\mathcal{F}_{\mathsf{GP2PC}}$, $\mathcal{S}$ receives $\text{UFIN}_S \oplus \text{E}_{13}$ from $\mathcal{A}$ for an error E_{13}.
11. On behalf of the server, $\mathcal{S}$ sends $(\text{H}_S, \text{FIN}_S)$ to $\mathcal{A}$ following the TLS specification. Then, on behalf of $\mathcal{P}$, $\mathcal{S}$ receives $(\text{H}_S, \text{FIN}_S \oplus \text{E}_{14})$ from $\mathcal{A}$ for an adversarial-chosen error E_{14}, and forwards it to the verifier controlled by $\mathcal{A}$. Given $(\mathsf{key}'_S, \mathsf{st}_S, \ell_S, \text{H}_S, \text{FIN}_S \oplus \text{E}_{14})$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the sub-protocol execution $\Pi^{(2)}_{\mathsf{AEAD}}$ towards type_1 = "decryption" and type_2 = "open". During this execution, $\mathcal{S}$ (on behalf of $\mathcal{P}$) checks the correctness of $\text{FIN}_S \oplus \text{E}_{14}$ and obtains $\text{UFIN}_S{}'$. Simulator $\mathcal{S}$ checks $\text{UFIN}_S \oplus \text{E}_{13} = \text{UFIN}_S{}'$. If any check fails, $\mathcal{S}$ aborts. $\mathcal{S}$ also obtains the shares of both parties on $\left[h^i_S\right]_{2^{128}}$ for $i \in [1, m]$. Then, $\mathcal{S}$ updates $\mathsf{st}_S := \mathsf{st}_S + 2$.
12. Given $(\mathsf{key}'_C, \mathsf{st}_C, \ell_Q, \text{H}_Q)$ and the shares of both parties on $\left[h^i_C\right]_{2^{128}}$ for $i \in [1, m]$, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the record phase of sub-protocol execution $\Pi^{(3)}_{\mathsf{AEAD}}$ with type_1 = "encryption"

and type_2 = "secret". During this execution, an AEAD ciphertext $(\mathbf{C}_Q, \sigma_Q) \oplus \mathrm{E}_{15}$ on the query is generated where $\mathbf{C}_Q$ is uniform and E_{15} is an error. Then, $\mathcal{S}$ sends $(\mathrm{H}_Q, \mathrm{ENC}_Q \oplus \mathrm{E}_{15})$ to $\mathcal{A}$ where $\mathrm{ENC}_Q = (\mathbf{C}_Q, \sigma_Q)$.

13. On behalf of the server, $\mathcal{S}$ receives $(\mathrm{H}_Q, \mathrm{ENC}_Q \oplus \mathrm{E}_{15} \oplus \delta)$ from $\mathcal{A}$ for an adversarial-chosen error δ, and then checks its correctness following the protocol specification and aborts if the check fails. Then, $\mathcal{S}$ samples $\mathbf{C}_R$ uniformly at random without knowing R, and then generates a GMAC tag σ_R with key_S following the AEAD specification, where key_S is computed by $\mathcal{S}$ via simulating the server honestly. Then, $\mathcal{S}$ sets $\mathrm{ENC}_R = (\mathbf{C}_R, \sigma_R)$ and sends $(\mathrm{H}_R, \mathrm{ENC}_R)$ to $\mathcal{A}$. On behalf of $\mathcal{P}$, $\mathcal{S}$ receives $(\mathrm{H}_R, \mathrm{ENC}_R \oplus \mathrm{E}_{16})$ from $\mathcal{A}$ for an error E_{16} chosen by $\mathcal{A}$.
14. $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{AEAD}}$ to simulate the post-record phase of sub-protocol executions $\Pi^{(1)}_{\mathsf{AEAD}}$, $\Pi^{(2)}_{\mathsf{AEAD}}$ and $\Pi^{(3)}_{\mathsf{AEAD}}$. During this execution, $\mathcal{S}$ makes $\mathcal{A}$ obtain $(h_C, h_S, \mathrm{z}_C, \mathrm{z}_S, \mathrm{z}_Q)$.
15. In the post-record phase, $\mathcal{S}$ emulates the (revealandprove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by receiving $\mathsf{pms}_{\mathcal{V}} \oplus \mathrm{E}_4$ from $\mathcal{A}$. In parallel, $\mathcal{S}$ receives $t_{\mathcal{V}} \in \mathbb{Z}_q$ from $\mathcal{A}$ and checks that $T_{\mathcal{V}} = t_{\mathcal{V}} \cdot G$. On behalf of $\mathcal{P}$, $\mathcal{S}$ performs the local check on all opened values and AEAD ciphertexts sent in the handshake phase (step 14) following the protocol specification. Then $\mathcal{S}$ checks that $\mathrm{E}_{15} = 0$ and $\mathrm{E}_{16} = 0$. If any check fails, $\mathcal{S}$ aborts.
16. $\mathcal{S}$ emulates the (prove) command of functionality $\mathcal{F}_{\mathsf{GP2PC}}$ by computing $\mathsf{pms}^* := \mathit{AddModp}(\mathsf{pms}_{\mathcal{P}} \oplus \mathrm{E}_5, \mathsf{pms}_{\mathcal{V}} \oplus \mathrm{E}_4)$.
17. Given pms^*, $\mathcal{S}$ invokes $\mathcal{S}_{\mathsf{PRF}}$ and $\mathcal{S}_{\mathsf{AEAD}}$ to simulate 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 $\Pi^{(3)}_{\mathsf{AEAD}}$. In the process, $\mathcal{S}$ obtains key^*_S as well as the local keys on $[\![Q]\!]$.
18. $\mathcal{S}$ emulates the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by computing $\mathrm{z}_R := \mathsf{AES}(\mathsf{key}^*_S, \mathsf{st}_S)$ and receiving the local keys on $[\![\mathrm{z}_R]\!]$ from $\mathcal{A}$. Then, $\mathcal{S}$ sends z_R to $\mathcal{A}$, and emulates the (check) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ on $[\![\mathrm{z}_R]\!] - \mathrm{z}_R$ by always sending true to $\mathcal{A}$.
19. $\mathcal{S}$ emulates the (zkauth) command of functionality $\mathcal{F}_{\mathsf{IZK}}$ by receiving the local keys on $[\![R]\!]$ from adversary $\mathcal{A}$.
20. $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{Conv}}$ to convert $([\![Q]\!], [\![R]\!])$ into additively homomorphic commitments by sending commitment identifiers to $\mathcal{A}$.

The following analysis builds upon the analyses of sub-protocols Π_{E2F}, Π_{PRF} and Π_{AEAD}. From the above simulation, it is clear that all functionalities emulated by $\mathcal{S}$ behave just like as that in the real protocol execution. If one of the errors E_1 and E_2 is not zero, then the protocol would abort, as the signature scheme is EUF-CMA secure. Therefore, the simulation of checking $\mathrm{REQ}_C \oplus \mathrm{E}_1$ and $\mathrm{RES}_S \oplus \mathrm{E}_2$ is computationally indistinguishable from that in the real protocol execution. In the following analysis, we always assume that the protocol execution does not abort. Otherwise, the simulation is natural to be indistinguishable from the real protocol execution.

The local check performed by $\mathcal{P}$ guarantees that all values obtained by $\mathcal{P}$ are computed correctly, and $\mathsf{pms} = \mathsf{pms}^* = \mathit{AddModp}(\mathsf{pms}_{\mathcal{P}} \oplus \mathrm{E}_5, \mathsf{pms}_{\mathcal{V}} \oplus \mathrm{E}_4)$ meaning that $\mathrm{E}_6 = 0$. Based on the analysis of sub-protocol Π_{E2F}, we obtain that both E_4 and E_5 are identical to zero. Besides, we have that all of $\mathrm{E}_7, \mathrm{E}_8, \mathrm{E}_9, \mathrm{E}_{10}, \mathrm{E}_{13}$ are equal to zero, due to the local check by $\mathcal{P}$. If $\mathrm{E}_3 \neq 0$, then $\mathcal{A}$ has to guess the value $\mathrm{FIN}_S = \mathsf{Func}_1(\mathsf{ms})$ before $\mathcal{P}$ receives the server finished message, where Func_1 is a function to compute a server finished message with a master secret ms based on PRF and AES. Combining the analysis of sub-protocol Π_{E2F} with that of sub-protocol PRF, $\mathsf{ms} \in \{0,1\}^{384}$ is computationally indistinguishable from a uniform string under the PRF-ODH assumption. Therefore, FIN_S is computationally indistinguishable from a random value in the ROM. Therefore, the probability that $\mathrm{E}_3 \neq 0$ is negligible.

Due to the local check on $\text{FIN}_C \oplus \text{E}_{11}$ performed by $\mathcal{P}$, we have that $\text{E}_{11} = 0$. Since $\text{E}_1, \text{E}_2, \text{E}_3$ are 0 with overwhelming probability based on the above analysis, we have that the client finished message is always computed correctly by the server. Therefore, $\text{FIN}_C \oplus \text{E}_{12}$ received by the server is identical to the client finished message computed with the correct pms, meaning that $\text{E}_{12} = 0$. From the local check on $\text{FIN}_S \oplus \text{E}_{14}$ performed by $\mathcal{P}$, we obtain that $\text{E}_{14} = 0$. In the real protocol execution, the correctness of ENC_Q is checked by $\mathcal{P}$ using the application key key_C and real query Q. However, $\mathcal{S}$ does not know the query Q and instead checks that the error $\text{E}_{15} = 0$. Based on the analysis of sub-protocol Π_{AEAD}, all of $\mathsf{key}_C, h_C, \text{Z}_Q$ are computationally indistinguishable from uniform strings and $\mathcal{A}$ cannot forge any GMAC tag with key_C in the record phase. Therefore, two methods to check ENC_Q are equivalent. Similarly, checking $\text{E}_{16} = 0$ is equivalent to checking ENC_R with key_S, where $\mathcal{A}$ cannot forge any GMAC tag with key_S. Therefore, the IT-MACs $(\llbracket Q \rrbracket, \llbracket R \rrbracket)$ commit to the consistent Q and R in the real protocol execution. Furthermore, by calling $\mathcal{F}_{\mathsf{Conv}}$, the values Q and R involved in additively homomorphic commitments are also consistent.

In the ideal-world execution, $\mathcal{S}$ generates the AES ciphertexts $\mathbf{C}_Q$ and $\mathbf{C}_R$ on the query Q and response R by sampling them uniformly. In the real protocol execution, Q and R are encrypted using the masks in the form of $\mathsf{AES}(\mathsf{key}, \mathsf{st}+i)$. We know that $\mathsf{key}_C, \mathsf{key}_S$ are computationally indistinguishable from uniform strings based on the analysis of sub-protocol Π_{AEAD}. Thus, in the ICM, $\mathbf{C}_Q$ and $\mathbf{C}_R$ are computationally indistinguishable from random strings in the real protocol execution. This means that the simulation of $\mathbf{C}_Q$ and $\mathbf{C}_R$ is indistinguishable from the real ciphertexts. Overall, the output of honest $\mathcal{P}$ and simulator $\mathcal{S}$ in the ideal-world execution is computationally indistinguishable from the output of honest $\mathcal{P}$ and adversary $\mathcal{A}$ in the real-world execution. $\square$