IEEE Copyright Notice

To appear in the Proceedings of 40th IEEE Symposium on Security and Privacy (S&P 2019), May 2019

Blind Certificate Authorities

Liang Wang * Gilad Asharov† Rafael Pass‡ Thomas Ristenpart§ abhi shelat¶

October 24, 2018

Abstract

We explore how to build a blind certificate authority (CA). Unlike conventional CAs, which learn the exact identity of those registering a public key, a blind CA can simultaneously validate an identity and provide a certificate binding a public key to it, without ever learning the identity. Blind CAs would therefore allow bootstrapping truly anonymous systems in which no party ever learns who participates. In this work we focus on constructing blind CAs that can bind an email address to a public key.

To do so, we first introduce secure channel injection (SCI) protocols. These allow one party (in our setting, the blind CA) to insert a private message into another party's encrypted communications. We construct an efficient SCI protocol for communications delivered over TLS, and use it to realize anonymous proofs of account ownership for SMTP servers. Combined with a zero-knowledge certificate signing protocol, we build the first blind CA that allows Alice to obtain a X.509 certificate binding her email address alice@domain.com to a public key of her choosing without ever revealing "alice" to the CA. We show experimentally that our system works with standard email server implementations as well as Gmail.

1 Introduction

Cryptography in practice relies on certificate authorities (CAs) that validate identities and provide a cryptographic assertion binding a public key to that identity. In addition to their use in systems like TLS, CAs are required in privacy-preserving or anonymous credential systems, first introduced by Chaum [16], subsequently studied extensively in the academic literature (c.f., [3,6,10,11,32,48]), and practically realized with systems like IBM's Identity Mixer [12] and Cinderella [18]. These systems have a user register with a CA (also called an identity provider) to obtain cryptographic credentials attesting to their identity or some attribute. The credential can then be used in an unlinkable, privacy-preserving way to subsequently authenticate with other systems.

But in all existing systems, registration reveals to the CA the identity of participants. This makes the CA a single point of privacy failure in settings where simply using a privacy tool is sensitive, such as journalists or dissidents living within repressive regimes, or whistleblowers at a large corporation. We call a system that does not disclose to any party the identities of participants as achieving participation privacy, and ask in this work whether it is possible to build participation-private systems that nevertheless utilize validated identities. A priori the answer would appear to be "no", because validating an identity would seem to fundamentally require knowing it.

*UW-Madison, liangw@cs.wisc.edu
†Cornell Tech, asharov@cornell.edu
‡Cornell Tech, rafael@cs.cornell.edu
§Cornell Tech, ristenpart@cornell.edu
¶Northeastern University, a.shelat@northeastern.edu

In this work we make progress on this question by designing the first ever *blind CA* for email identities. Our blind CA validates ownership of an email address and issues a credential binding that email address to a public key, but never learns the email address being used. What's more, our system achieves this in a legacy-compatible way, utilizing existing email systems and producing X.509 certificates. By combining our blind CA with Cinderella [18], one can achieve the first anonymous credential system achieving participation privacy.

The main challenge involves the tension between the need to validate an identity while not learning it. We resolve this tension using what we call an anonymous proof of account ownership (PAO). Consider an email provider, such as Gmail, a verifier (the blind CA), and the prover that owns an email account with the provider. To achieve participation privacy, the verifier should be able to validate ownership of the account by the prover, without the prover revealing which account and without the email provider learning that the prover is participating.

To do this, we introduce a more general tool called secure channel injection (SCI). An SCI protocol allows the prover and verifier to jointly generate a sequence of encrypted messages sent to a server, with the ability of the verifier to insert a private message at a designated point in the sequence. In our proof of ownership context, the server is run by the email provider, and the injected message will be a random challenge inserted into an email. To complete the proof of ownership, the prover can later retrieve the challenge from the service using a separate connection, and send the challenge back to the verifier.

Our SCI construction targets protocols running over TLS, which is the most widely used secure channel protocol. Recall that TLS consists of a handshake that establishes a shared secret, and then encrypts application-layer messages (SMTP in our context) using a record layer protocol that uses an authenticated encryption scheme. We design efficient, special-purpose secure two-party computation protocols that allow the prover and verifier to efficiently compute a TLS session with the server. For most of the session, the verifier acts as a simple TCP-layer proxy that forwards messages back and forth. The prover negotiates a TLS session key directly with the destination server. At some point in the stream, however, the verifier must inject a message, and here the prover (which has the session key) and the verifier (which has the secret challenge to inject) perform an interactive protocol to compute the record layer encryption of the message. By exploiting the cryptographic structure of the TLS record layer encryption scheme, we securely achieve this using a protocol whose most expensive step is a two-party secure computation protocol [60] on a circuit consisting of a small number of AES computations (plus exclusive-or operations). Whereas direct use of secure computation to perform the entire record layer construction would be expensive, our approach is demonstrably feasible and leverages recent advances in two-party secure computation of AES. Unlike Multi-context TLS [44], our protocol can modify TLS sessions in a fully legacy-compatible way, without changing existing network infrastructures and collaborations with network providers.

Our SCI-based anonymous PAO protocol can additionally output a cryptographic commitment to the prover's identity (the email account name). The prover can then construct an X.509 certificate for a public key of their choosing, hash it, and prove in zero-knowledge to the verifier that the identity field of the X.509 matches the email account name in the commitment. We use the ZKBoo framework for this step [27]. If the proof verifies, then the verifier obliviously signs the hash. In this way, the verifier never learns the identity but provides the certificate only should the prover have a valid email account with the agreed-upon service.

We provide formal analysis of the protocols underlying our blind CA, showing security holds even for malicious provers or malicious verifiers, and for honest-but-curious email services. Security of the final blind CA protocol relies on some nuances of SMTP implementations, which we discuss in the body and verify empirically.

We implement a prototype and test it with various SMTP servers, showing that it is fast enough for deployment. (We plan to make our implementation public and open source.) Running the prover on a laptop connected via a public wireless network to a verifier running on EC2, the median time to complete an SCI-based anonymous PAO is 760 milliseconds. More performance results are given in the body.

In summary, our contributions include the following:

- We introduce the notion of secure channel injection and show how to realize it efficiently in the case of TLS. Our techniques can be adapted to other secure channels such as SSH and IPsec.
- We use SCI to build anonymous proof of account ownership protocols for SMTP over TLS.
- We show how to combine all this to construct a blind CA that generates certificates binding a public key to an account after verifying ownership of the account, all without having the CA learn which account was used.

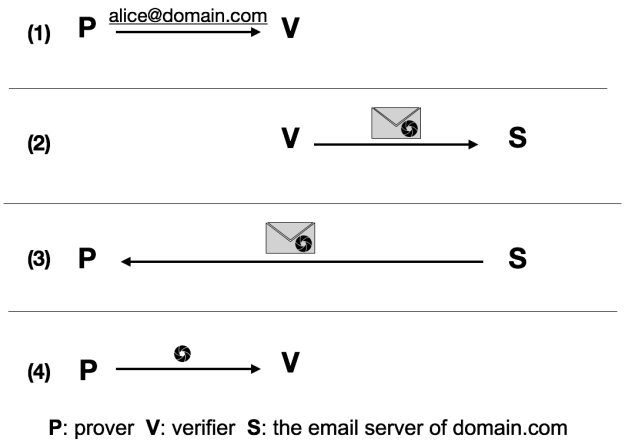
By combining our blind CA with Cinderella, we achieve the first participation-private anonymous credential system. Our results therefore provide a foundation for privacy tools in contexts where revealing usage of privacy tools can be dangerous. Finally, we note that there is nothing fundamental about our use of email for identities. Future work could use our techniques to build blind CAs for other types of identities, e.g., accounts on popular web services.

2 Background and Overview

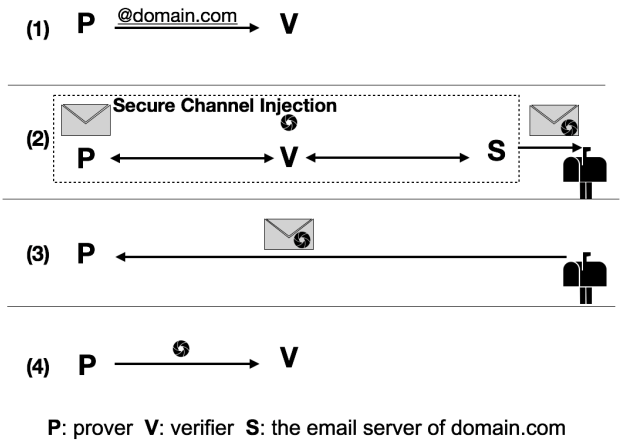
We show how to build a blind CA service that can verify a user's ownership of an account, and then sign an X.509 certificate binding the user's public key to the account—without the CA learning the account or public key of the user. Our blind CA is based on an anonymous proof-of-account ownership (PAO) for email. We proceed with an overview of a standard PAO, and then with a high-level idea underlying our construction of an anonymous PAO.

Proofs of account ownership. Proofs of email ownership are a primary form of authentication on the web today and form a backstop in case of loss of other credentials (e.g., a forgotten password). A conventional proof of email ownership works as follows. The alleged owner of an email address, say `alice@domain.com`, is who we will refer to as the prover. The prover tells a verifier her email address, and in response the verifier challenges her by sending to `alice@domain.com` an email containing a random, unpredictable challenge. The prover must recover this challenge and submit it back to the verifier. If successful, the verifier is convinced that the prover can, indeed, access the account and presumably owns it. (Of course it could be anyone with access to the email account, including rogue insider admins or those who have compromised the account credentials.) See Figure 1 for an illustration. Email is one example of a broader class of account ownership challenge-response protocols. Ownership of a domain name is often proven by having the owner set a field of the DNS record to a challenge value supplied by a verifier. Ownership of websites can be proven by adding a webpage that contains a challenge value, and similar approaches work with Twitter and Facebook accounts [35]. Common to all proofs of account ownership is the fact that the verifier learns the identity of the prover.

Public-key registration. Proofs of account ownership have become increasingly used by certificate authorities (CAs) to verify ownership of an identity when registering a public key in a public-key infrastructure (PKI). One example is the Let's Encrypt service [34], which provides free TLS certificates to users that can prove ownership of the domain via a DNS proof of ownership or web



(a) Proof of account ownership:
(1) The prover wishes to prove the verifier that she is the owner of `alice@domain.com`; (2) The verifier interacts with the service `domain.com`, and sends an email to `alice@domain.com` with some unpredictable challenge; (3) The prover accesses her account at `domain.com` and extracts the challenge; (4) The prover sends the challenge to the verifier, proving ownership of the account `alice@domain.com`.



(b) Anonymous proof of account ownership:
(1) The prover wishes to prove the verifier that she is some eligible user of `domain.com` without revealing her identity; (2) Secure channel injection: the prover sends an email from `alice@domain.com` to some other email account she owns; the verifier sees that the interaction is with `domain.com` and injects a challenge into this email at some designated point; (3) The prover accesses her other email account and extracts the challenge; (4) The prover sends the challenge to the verifier, proving ownership of some account in `domain.com`.

Figure 1: A regular proof of account ownership versus our anonymous proof of account ownership. P is the prover, S is the email server of `domain.com`, and V is the verifier. The black circles represent challenges.

page proof of ownership. Keybase.io signs PGP keys based on proofs of ownership of social media accounts [35]. Traditional CAs also need to do PAOs, e.g., proof ownership of the administrative email of the domain, to validate one's ownership of a domain, before issuing a certificate binding a public key to the domain. In these contexts, the user sends her identity and public key to the CA, the latter invokes a proof of ownership protocol, and if the proof verifies then the CA provides the user with an appropriate certificate. Importantly, the CA in all existing systems learns the identity of the user.

Anonymous proofs of account ownership. The conventional protocols discussed so far reveal to the verifier the identity of the account owner. Sometimes revealing the specific identity is important for security, for example if one needs to log users and detect fraudulent requests. But in some settings the provers may be unwilling to reveal their identities. In the end-to-end encryption setting, privacy is an often mentioned critique of certificate transparency mechanisms like CONIKS [42]. Existing anonymous credential systems might seem to solve this problem, but in fact current systems rely on a trusted third party (TTP) to perform identity checks (via conventional PAOs) and distribute pseudonyms to users. The pseudonym can be used to request a certificate from a CA, who checks the legitimacy of the pseudonym with the TTP [6, 12, 32, 48, 49]. However, these systems are vulnerable if the TTP misbehaves.

We show how to obtain an *anonymous proof of account ownership*. In our setting, we prove ownership not by showing the ability to read an email from the account, but rather by sending an email from the account. An illustration appears in Figure 1. In more detail, a prover with an email account `alice@domain.com` wishes to prove to some verifier that she owns an account at the domain without revealing her identity. The prover would authenticate to her account at that domain, and will send an email from her account to some other email account she owns at some

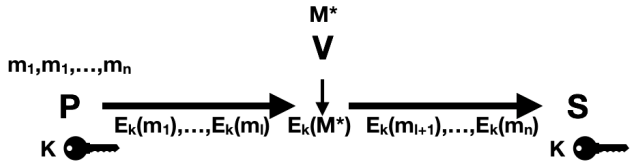


Figure 2: Secure channel injection: The prover (P) interacts with the service (S) while all the interaction is performed through the proxy of the verifier (V). At some designated point in the interaction, the proxy injects a secret message into the encrypted stream (without knowing the secret key K). This is done using a secure protocol between the prover and the verifier, while the server is unaware of this injection.

other domain (or even to the same email address, `alice@domain.com`). However, all communication between the prover and the domain would be performed via a proxy, which would be the verifier. Using secure computation techniques, we show how to allow the verifier to inject a secret challenge at some designated point into this encrypted connection. We call this subprotocol *secure channel injection*. Our techniques guarantee that the prover has no information about the secret challenge, and the only way to recover it is by accessing the recipient email account. To prove ownership, the prover accesses her other email account, extracts the challenge and presents it to the verifier. A diagram of secure-channel injection appears in Figure 2.

Secure channel injection. To build an anonymous PAO and blind CA, we will develop an underlying primitive that we refer to as secure channel injection (SCI). The idea is to allow a party to inject a (relatively) small amount of information into a secure connection between a client and server. In the ownership proof context, the client will be the prover, the server will be the authenticated service, and the verifier will be the party injecting data. In our realizations the latter will end up being a specialized proxy that relays traffic between the prover and the service. While we explore use of SCI protocols in the context of anonymous PAOs, future work might surface other applications.

General tools for secure computation enable computing SCI for any ciphersuite of TLS, but could be expensive. We demonstrate *efficient* realization of secure channel injections for TLS with two ciphersuites: (1) TLS with cipher block chaining mode of operation (AES-CBC) with HMAC-SHA256 authentication, and (2) AES with Galois counter mode (AES-GCM). In the first case, we construct a protocol whose most expensive step is a two-party secure computation protocol [60] on a circuit consisting of a small number of AES computations (plus exclusive-or operations). Our approach is demonstrably feasible and leverages all of the recent advancements in two-party secure protocol construction for computing AES. In the second case we construct a protocol which the only expensive operation is an oblivious polynomial evaluation [26, 29, 30, 43] needed for computing the authentication data, and no secure computation of AES is necessary. We prove security of our protocols in the random oracle model.

In both cases the role of the proxy is constrained — our protocols ensure that even an actively malicious proxy cannot mount an arbitrary man-in-the-middle attack, but only are able to insert a constrained amount of data.

In the next couple of sections we go through the details of our approach. We first present secure-channel injection protocols (§3) and how to realize them for TLS (§4). We then show how to use SCI to build anonymous PAOs for email and, ultimately, our blind CA for email (§5).

Use cases. For concreteness, we provide example use cases for which blind CAs may be useful:

(1) Consider when the organization is a bank and the prover is a whistleblower that must prove to a reporter her status as an insider without revealing her identity. The bank is unlikely to aid the user by setting up an anonymous credential system. Using blind CA, the bank is unaware that it is being used as an identity provider.

(2) Cinderella, an anonymous credential system that can perform X.509 certificate verification via zero-knowledge proofs, can be used with anonymous voting services to hide voters' identities (i.e., subject id in the X.509 certificate) [18]. However, it assumes each voter already has a unique X.509-compatible, personal certificate, and uses a voter's public key and other public information to generate a pseudonym for the voter. Once CAs cooperate with the voting services, it's easy to recover the true identity of a voter under a given pseudonym. If one used a blind CA instead, then identities are never learned, let alone collected, by the CA (assuming no collusion with the email service).

3 Secure Channel Injection

A secure channel injection (SCI) protocol is a three-party protocol between a client, a proxy, and a server, parameterized by a message template $M_t = (|M_t^p|, |M^*|, |M_t^s|)$, which can be thought of as a "placeholder" for actual messages. The client holds as input a message prefix $M_t^p \in \{0,1\}^{|M_t^p|}$ and a message suffix $M_t^s \in \{0,1\}^{|M_t^s|}$, communicates with the server, where the proxy is interested in "injecting" a random message/challenge $M^* \in \{0,1\}^{|M^*|}$ into that interaction. We follow the standard definition for secure computation in the malicious adversarial model (e.g., [13, 28]). The secure channel injection protocol computes Functionality 1.

Functionality 1: Message Injection (parameterized with a message template: $M_t = (|M_t^p|, |M^*|, |M_t^s|)$)

- **Input:** The client holds some input prefix message $M_t^p \in \{0,1\}^{|M_t^p|}$ and suffix $M_t^s \in \{0,1\}^{|M_t^s|}$. The proxy holds some message $M^* \in \{0,1\}^{|M^*|}$ that was chosen from some high-entropy source. The server has no input.
- **Output:** The server outputs (M_t^p, M^*, M_t^s). The proxy and the client have no output.

The following simple protocol computes the aforementioned functionality: The proxy chooses a message M^* uniformly at random, and both client and proxy just transmit their messages to the server. However, we are interested in protocols that compute this functionality but also satisfy the following two properties:

- First, we are interested in protocols where the code of the server is already "fixed", and the messages that the client and the proxy send to the server must match some specific syntax. Specifically, the protocol is additionally parameterized by a secure channel protocol $\mathcal{SC} = (\mathcal{SC}_{Cl}, \mathcal{SC}_S)$. The code of the server in the secure channel injected protocol is fixed to $\mathcal{SC}_S$,[1] modeling the fact that the proxy has to "inject" a message into an existing secure channel communication $\mathcal{SC}$ between the client and the server.
- Second, the network setting is such that there is no direct communication channel between the client and the server. All messages between these parties are delivered through the proxy. In particular, this already requires the client to encrypt its messages as the proxy should not learn any information about M_t^p, M_t^s (besides the already known message template, i.e., known sizes).

[1]The only exception is that, in order to have a meaningful definition of the problem, we change the code of the server, letting it output the decrypted transmitted messages $M_1, \ldots, M_m$ in case of successful authentication. If the authentication fails, it outputs $\perp$.

For the purposes of description it suffices to give a simplified view of secure channel (SC) protocols. Let $\mathcal{SC} = (\mathcal{SC}_{Cl}, \mathcal{SC}_S)$ consist of a key exchange phase followed by transmitting r ciphertexts $E(K, M_1), \ldots, E(K, M_r)$ from the client to the server encrypted under a symmetric encryption algorithm E and session key K, regardless of whether it is stateful or not. Recall that we follow the standard definition for secure computation in the malicious adversarial model (e.g., [13, 28]), for analyzing whether a protocol securely realizes a given functionality. This leads us to the following definition of a secure channel injection:

Definition 2 (Secure Channel Injection)**.** *Let $\mathcal{SC} = (\mathcal{SC}_{Cl}, \mathcal{SC}_S)$ be a secure channel protocol between a client and a server. We say that a three party protocol $\mathcal{SCI} = (\Pi_{Cl}, \Pi_{Pr}, \Pi_S)$ is a* secure channel injection protocol for *$\mathcal{SC}$, if the following conditions hold: (1) $\mathcal{SCI}$ securely realizes Functionality 1, and (2) $\Pi_S = \mathcal{SC}_S$.*

Security properties. As for condition (1) in Definition 2, we require that security holds when either the client or the proxy is malicious (meaning, it can deviate arbitrarily from the protocol specification), or the server is an honest-but-curious adversary, meaning it might try to violate security by inspecting the sequence of packets sent to and from it and the sequence of plaintext messages, but it won't maliciously deviate from the protocol specification. Our definition guarantees the following basic security goals:

(1) *Injection secrecy*: The client cannot learn M^* during the protocol interaction.[2]

(2) *Transcript privacy*: The proxy does not learn anything about messages other than M^*.

(3) *Transcript integrity*: The proxy should not be able to modify parts of the message transcript besides M^*.

(4) *Server obliviousness*: Condition (2) in Definition 2 guarantees that the server cannot distinguish an SCI execution from a standard execution of the underlying $\mathcal{SC}$ protocol with the client.

We assume that the IP address of the proxy does not, by itself, suffice to violate server obliviousness. We emphasize that as opposed to the server (which is oblivious to the fact that it is not participating in a standard execution of the underlying $\mathcal{SC}$ protocol with the client), the client is well-aware that this is not a standard execution. In fact, the client intentionally collaborates with the proxy in order to enable it to inject the secret message M^*, and the two parties together compute a valid record that contains the injected message. The client and the proxy together can therefore be viewed as a single unified client interacting with the server in a standard secure channel protocol.

Network assumptions. As mentioned before, we assume that the client and the server cannot communicate directly, and their communication is delivered through the proxy. We also assume each party can only observe their local network traffic. That is, the server cannot access the network transcripts between the client and the proxy (otherwise, we can never achieve server obliviousness), and the client cannot access the network transcripts between the proxy and the server (otherwise, we cannot simultaneously achieve transcript injection secrecy with server obliviousness).

We will not consider attackers who are capable of manipulating network routing and injecting spoofed packets. This rules that the situation that the client bypasses the proxy and spoofs and session between the proxy and the server. Other types of attacks that are not directly related to

[2] In our applications of SCI, the client will eventually learn M^* by retrieving it later from the server. But it should not be learned before.

the goals of adversaries as mentioned above, such as denial-of-service attacks, are not taken into account. We also don't yet consider implementation-specific attacks such as vulnerabilities in the proxy software.

Relaxations of Functionality 1. For conceptual simplicity, we presented Functionality 1 for the most simplified settings. In order to design more efficient protocols, however, a somewhat more complicated functionality is necessary. In a nutshell, the modifications allow the proxy to learn some leakage on M_t^p, M_t^s (such as known headers or part of the messages the client does not have to hide). For the AES-GCM SCI protocol (see Appendix .2) the ideal functionality additional allows the client to "shift" the injected message M^* by sending the trusted party some message Δ (and letting the output of the server be $(M_t^p, M^* + \Delta, M_t^s)$). This suffices for our application as this functionality satisfies injection secrecy.

We now turn to design cryptographic protocols for which we can prove that they realize the SCI ideal functionality (without relying on any trusted third party).

4 SCI for TLS

We focus in this paper on TLS as the secure channel. Using common MPC techniques, such as Yao's protocol [60] or fully homomorphic encryption [25], every secure channel protocol can be converted into an SCI. However, these general techniques would be expensive due to the complexity of the TLS record construction, which involves computations for HMAC, AES, and record padding. We can do better by taking advantage of the way TLS encryption works. While there are several options supported in the wild, we focus on the currently commonly used ones:

(1) AES using CBC with HMAC-SHA-256. This mode is widely used in TLS 1.1 and 1.2, and we show how to build for it an SCI in Section 4.1. Our protocol requires general-purpose MPC on just a few invocations of AES, making it fast.

(2) AES with Galois / Counter mode (AES-GCM). We provide the SCI protocol in Appendix .2. Our protocol relies on oblivious polynomial evaluation.

In the sequel, we focus on AES using CBC with HMAC-SHA-256 (Section 4.1).

4.1 TLS with AES-CBC and HMAC-SHA256

The TLS record is described in Figure 3. We need to design a protocol that allows two parties to jointly compute such a TLS record, where the client provides the TLS session keys and part of the message, and the proxy provides the injected message. We first introduce two sub-protocols called 2P-HMAC and 2P-CBC, and design SCI-TLS based on the two protocols. In a nutshell, 2P-HMAC boils down to submitting two partial tags, one from the client to the proxy and one from the proxy to the client, and its overhead is minor. In 2P-CBC, the client computes the AES ciphertexts of M_t^p, M_t^s locally, and the parties engage in a secure protocol for computing AES on the blocks of M^* (where the proxy inputs the blocks of M^* and the client inputs the key, i.e., we realize an oblivious PRF). We fully specify the protocols and then analyze the security of SCI-TLS. We remark that we do not formalize the ideal counterparts of 2P-HMAC and 2P-CBC, and we analyze the security of SCI-TLS protocol as a whole. We divide the protocol into these subprotocols just for expositional clarity.

Assume the client Cl holds keys K_{hmac} and K_{aes} as well as an injection template prefix M_t^p and suffix M_t^s. A proxy holds the injected message M^*. We will show how they can jointly compute

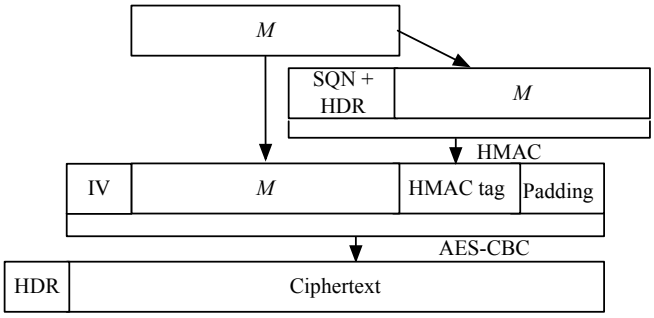


Figure 3: The MAC-then-encrypt construction in TLS (version $\geqslant$ 1.1). HDR is a 40-bit TLS record header and SQN is a 64-bit sequence number. IV has a fixed size of 128-bits. The size of the HMAC tag depends on the hash functions being used in HMAC. Before AES encryption, the record will be padded to a multiple of 128 bits [21, 22].

HMAC with the first key over $M_t^p \| M^* \| M_t^s$ and CBC mode with the second key over the same composed message. We denote the HMAC chunk size by d (in bits), and assume CBC mode uses a blockcipher whose block size in bits we denote by n. Looking ahead, we will require that M_t^p and M^* each have length a multiple of d (after headers are prepended) during the HMAC computation and n during CBC.

2P-HMAC. Recall that HMAC is a pseudorandom function (PRF) constructed on top of a hash function that we denote H. We assume that H is a Merkle-Damgård based hash function, which aligns with the hashes used in TLS.[3] We take advantage of the fact that one can outsource computation of HMAC over portions of messages without revealing other parts of the message or the key.

Let $f : \{0,1\}^v \times \{0,1\}^d \to \{0,1\}^v$ be the compression function underlying H. It accepts messages of length d bits and a string called the chaining variable of length v bits. It outputs a v-bit value. For any string $S \in \{0,1\}^v$ and string $M = M_1, \ldots, M_m$ where each M_i is d bits long, we let $f^+(S, M)$ be defined recursively by $S_i = f(S_{i-1}, M_i)$ for $i = 1$ to m and $S_0 = S$. Finally $f^+(S, M) = S_m$. For the hash functions of interest one appends to a message M a padding string $\mathrm{PadH}_{|M|}$ so that $M \| \mathrm{PadH}_{|M|}$ is a multiple of d bits. For SHA-256 for example $\mathrm{PadH}_\ell = 10^r \| \langle \ell \rangle_{64}$ where the last part is a 64-bit encoding of ℓ and r is defined to produce enough zeros to make $\ell + r + 65$ a multiple of d. Finally the full hash is defined as $H(M) = f^+(\mathrm{IV}, M \| \mathrm{PadH}_{|M|})$.

HMAC on a key K and message M is built using H as follows:

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

where *ipad* and *opad* are the inner and outer padding constants each of length d bits [38]. In our usage $|K| < d$, so one first pads it with zero bits to get a d-bit string before applying the pad constants. To perform a joint computation of $\mathsf{HMAC}(K_{hmac}, M_t^p \| M^* \| M_t^s)$ the parties follow the protocol detailed in Protocol 3. We denote an execution of this protocol by $\mathsf{2P\text{-}HMAC}((K_{hmac}, M_t^p, M_t^s), M^*)$.

2P-CBC. We now turn to how to jointly compute a CBC encryption over $M_t^p \| M^* \| M_t^s$. Since we are working now with n-bit strings, we let $M_t^p = (P_1, \ldots, P_q)$, $M^* = (P_{q+1}, \ldots, P_{q+r})$, and $M_t^s = (P_{q+r+1}, \ldots, P_t)$, where each P_i is an n-bit block.

The CBC mode on message $M = (P_1, \ldots, P_t)$ is defined by choosing a random n-bit $C_0 = IV$ and computing $C_i = \mathsf{AES}_{K_{aes}}(C_{i-1}, P_i)$ for every $i = 1, \ldots, t$, and outputting $C_0, \ldots, C_t$. If P_t is not a multiple of n, then some $\mathrm{PadC}_{|M|}$ is added to the message to ensure that M is a multiple of n bits

[3] Our protocol here will not work with SHA-3 whose compression function is not secure (e.g, Keccak, who uses a sponge construction [7]). This is related to so-called mid-game attacks [14].

Protocol 3: **2P-HMAC**$((K_{hmac}, M_t^p, M_t^s), M^*)$
Input: The client holds K_{hmac}, $M_t^p = (M_1, \ldots, M_\ell)$, and $M_t^s = (M_{\ell+k+1}, \ldots, M_m)$. The proxy holds $M^* = (M_{\ell+1}, \ldots, M_{\ell+k})$ where each $M_i \in \{0,1\}^d$.
The protocol:

(1) The client computes $s_0 = f(IV, K_{hmac} \oplus ipad)$, and for every $i = 1, \ldots, \ell$, it computes $s_i = f(s_{i-1}, M_i)$.
Send s_ℓ to the proxy.

(2) The proxy computes $s_i = f(s_{i-1}, M_i)$ for $i = \ell+1, \ldots, \ell+k$. Send $s_{\ell+k}$ to the client.

(3) The client proceeds $s_i = f(s_{i-1}, M_i)$ for all $i = \ell+k+1, \ldots, m$, and then $s^* = f(s_m, \mathsf{PadH}_{|M|}) = H((K_{hmac} \oplus opad)||M)$.

Output: The client outputs $T = H((K_{hmac} \oplus opad)\|s^*)$.

in length. Our 2P-CBC protocol is described in Protocol 4. In order to compute the ciphertexts, the two parties use a general-purpose MPC protocol to compute $\mathsf{AES}_{K_{aes}}(P_i')$ where the client inputs K_{aes}, the proxy inputs some block P_i', and receives ciphertext C_i. We denote this functionality as F_{AES}.

Protocol 4: **2P-CBC**$((K_{aes}, M_t^p, M_t^s), M^*)$ (in the F_{AES}-hybrid model)
Input: The client holds K_{aes}, messages $M_t^p = (P_1, \ldots, P_q)$, and $M_t^s = (P_{q+r+1}, \ldots, P_t)$, the proxy holds $M^* = (P_{q+1}, \ldots, P_{q+r})$, where each $P_i \in \{0,1\}^n$.
The protocol:

(1) The client sets $C_0 = IV$, and computes $C_i = \mathsf{AES}_{K_{aes}}(C_{i-1} \oplus P_i)$ for every $i = 1, \ldots, q$. It sends $C_0, \ldots, C_q$ to the proxy.

(2) For $i = q+1, \ldots, q+r$, the client and the proxy F_{AES}-functionality for computing $\mathsf{AES}_{K_{aes}}(C_{i-1} \oplus P_i)$, where the client inputs the key and the proxy inputs the message. The proxy receives as outputs $C_{q+1}, \ldots, C_{q+r}$, and sends C_{q+r} to the client.

(3) The client proceeds to compute $C_i = \mathsf{AES}_{K_{aes}}(C_{i-1} \oplus M_i)$ for every $i = q+r+1, \ldots, t$ and sends all the ciphertext to the proxy.

Output: The proxy outputs $C_0, \ldots, C_t$.

We assume that $|M^*| \geqslant 2n$ (i.e., $r \geqslant 2$). If $r = 1$, the proxy cannot send C_{p+1} back to the client because the client can easily recover M^* based on her knowledge of C_p and K_{aes}. In this case, we can alternatively require that $|M_t^s| = 0$; that is, $|M^*|$ is the last block of the plaintext.

The SCI protocol. We are now in a position to describe our solution for SCI with TLS where the proxy wants to inject a message at some designated point into the stream of encrypted client-to-server message data. Let $Q_1, \ldots, Q_u^*, \ldots, Q_v$ be the sequence of TLS plaintext fragments sent from the client to the server in separate record layer encryptions, with Q_u^* representing the fragment within which the proxy will inject its private message M^*.

Recall that HMAC-SHA256 works on blocks of size $d = 512$ bits (64 bytes) and AES is on blocks of size $n = 128$ bits. Moreover, we recall that *SQN* and *HDR* (of total length $40 + 64 = 104$ bits) should be added to the message when computing the HMAC, whereas these are not included when encrypting with CBC (see Figure 3).

We consider the simpler case in which $|M^*| = 256$ bits (i.e., $|M^*| = 2n$, as in Protocol 4). We let $M_t^p = (M_t^{p_1}, M_t^{p_2})$ where $|M_t^{p_1}| = 408$ and $|M_t^{p_2}| = 232$ bits. Moreover, we let $M_t^s = (M_t^{s_1}, M_t^{s_2})$

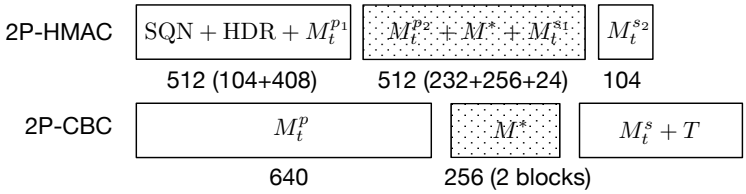


Figure 4: An example of injecting a 256-bit M^*. The messages with dots are input by the proxy, and the other (portions of) messages are provided by the client. The numbers are message sizes in bits.

where $|M_t^{s_1}| = 24$ bits and $M_t^{s_2} = 104$ bits. The client sends $M_t^{p_2}$ and $M_t^{s_1}$ to the proxy[4]. As such, the blocks $(SQN, HDR, M_t^{p_1})$ and $(M_t^{p_2}, M^*, M_t^{s_1})$ are each multiplies of d and $|M_t^p| = |(M_t^{p_1}, M_t^{p_2})|$ and $|M^*|$are each multiples of n. See Figure 4.

SCI-TLS proceeds by having the proxy act as a TCP-layer proxy for the TLS handshake between the client and the server and for the first $u-1$ TLS record layer fragments. Let the client-to-server session keys be K_{hmac} for HMAC and K_{aes} for AES. To send Q_u^* the client constructs the message prefix $SQN\|HDR\|M_t^{p_1}$. Then the client and the proxy execute $\mathsf{2P\text{-}HMAC}((K_{hmac}, SQN\|HDR\|M_t^{p_1}, M_t^{s_2}), M_t^{p_2}\|M^*\|M_t^{s_1})$ to compute the HMAC tag T. Next, they execute $\mathsf{2P\text{-}CBC}((K_{aes}, M_t^p, M_t^s\|T), M^*)$ to jointly compute the record layer ciphertext if $|M^*|$ is greater than 128 bits (16 bytes).

A special case is when the proxy wants to inject less than 256 bits. The minimal amount that our approach allows is 152 bits. This case somewhat corresponds to the $r = 1$ setting in Protocol 4, for which, as we mentioned before, the proxy cannot send back the ciphertext to the client. We can handle this case if $|M_t^s| = 0$, as we elaborate in the Appendix .3.

We model the internal function f of the hash function H as a random oracle, and AES as an ideal cipher, and prove the following Theorem in Appendix .3.1:

Theorem 5. *The above protocol is a secure channel injection protocol for TLS with* AES-CBC *and HMAC-SHA-256 (i.e., satisfies Definition 2), assuming that f is a random oracle and* AES *is an ideal cipher.*

AES-GCM. In Appendix .2 we demonstrate how to implement an efficient secure computation protocol for AES-GCM. In CBC mode with HMAC, we had a minor overhead for jointly computing the authentication tag and the expensive part was the joint computation of ciphertexts corresponding to M^*. In AES-GCM, the situation is the opposite. Here, the client chooses a random IV and uses counter mode, namely, all messages $M_1, \ldots, M_t$ are encrypted using the "key stream" $\mathsf{AES}_K(IV + 1), \ldots, \mathsf{AES}_K(IV + t)$. The client can simply send the portion of the key stream that is associated with the injected message of the proxy, and no secure computation of AES is needed. However, the authentication data involves evaluation of a polynomial that is not known to the client, on a point that is known only to the client. We therefore use oblivious polynomial evaluation [29, 30, 43] to perform this part of the computation. See full details in Appendix .2.

Other secure channels. We focused above on TLS using common record layer encryption schemes. Our techniques can be adapted to some other protocols and authenticated encryption schemes, such as Encrypt-then-MAC (e.g., IPsec) and Encrypt-and-MAC (e.g., SSH). There are a number of in-use authenticated encryption schemes such as ChaCha20/Poly1305 [47] and CCM [41] for which we have not yet explored how to perform efficient SCI. Common to them is the use of Encrypt-then-MAC type modes with a "weaker" MAC such as CBC-MAC. Theoretically, constructing SCI for these

[4]That is, this is the leakage the proxy receives regarding the input of the client.

secure channels is always possible. We leave constructing efficient SCI protocols for these schemes to future work.

5 Anonymous PAOs and Blind Certificate Authorities

We introduce two applications of SCI in this section: anonymous proofs of account ownership (PAOs) and blind certificate authorities (blind CAs). We focus on SMTP as the application layer protocol. We introduce a basic SMTP-STARTTLS workflow, our requirements of SMTP implementations, and our application design, and briefly discuss potential application-specific attacks and the corresponding defenses towards the end of this section.

5.1 SMTP with STARTTLS

We first briefly discuss the workflow of sending an email in SMTP-STARTTLS as our application is intimately tied to the workings of SMTP implementations. We focus on PLAIN as the target authentication mechanism, which is the most widely used authentication mechanism [33]. The client first sends a `STARTTLS` command to initialize a TLS-protected SMTP session after checking the SMTP server's support for TLS. In the session, the client sends the following commands in order: `AUTH PLAIN account_name password` (authentication), `MAIL` and `RCPT` (setting the sender/recipient addresses), `DATA` (notifying the begin of email transactions), the email content, and finally `QUIT` (closing the session). The `AUTH`, `MAIL`, `RCPT`, and `DATA` are mandatory and must be sent in order according to RFC [37], while other commands are optional. The client needs to wait for the server's response to a command before sending the next one, unless the `PIPELINING` extension, which allows the client to send several commands in a batch, is enabled by the server. But according to RFC [37], the server should respond to each command individually.

5.2 System Assumptions

The security of anonymous PAOs and blind CAs rely on the security of SCI. So, they have the same threat model and underlying assumptions as SCI (e.g., parties do not collude). See §3 and §4. Besides, we assume that the challenges generated during anonymous PAOs will not be leaked by the prover intentionally or unintentionally. And, the certificate generated by the blind CA should contain sufficient entropy to rule out brute-force inversion of the certificate hash, which is revealed to the CA. This requirement is easily satisfied as long as the certificate includes the prover's public key. We also assume that the prover might communicate with the verifier (the CA) through an anonymous or pseudonymous channel, such as Tor and public wireless networks, to achieve IP anonymity. The verifier must be associated with a valid certificate that can be used to identify the verifier.

We assume the target STMP server supports the following property: during one SMTP session, only one email, with a secret challenge injected in the designated location, will be generated and sent from the authenticated account, and each TLS message sent from the prover contains exactly one SMTP command as specified by our protocols. Thus, the SMTP server should meet the following requirements:

(1) Auth: The server should use correctly configured SMTPS (e.g., using valid TLS certificate and being configured as a closed relay) and only authenticated users with correct sender addresses can send emails [23, 40, 59]. (2) NoEcho: the server should not echo back received commands to the client, which would immediately break injection secrecy. (3) NoPipeline: the server should not support the `PIPELINING` extension; or (4) RFCCompliant: If the server does not satisfy NoPipeline,

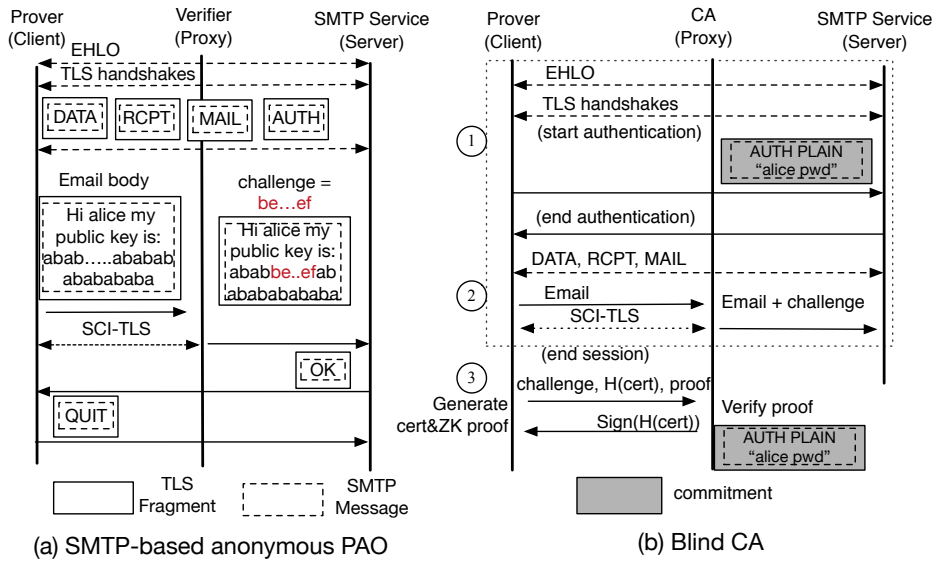


Figure 5: Technical flows of (a) SMTP-based anonymous PAO and (b) blind CA. For blind CA: ① The CA saves the first message as the commitment; ② The CA and the prover use anonymous PAO to inject a challenge; ③ The CA and the prover run a zero-knowledge protocol to generate a legitimate X.509 certificate.

the server must be RFC-compliant in terms of how it responds to pipelined commands, as described in §5.1. These requirements help anonymous PAOs and blind CAs to achieve security (See §5.5).

To understand how stringent these requirements are, we investigated the behavior of 150 popular SMTP servers (supporting STARTTLS) from public lists available on the Internet [2, 24]. Overall, we found 112 SMTP servers (75% of those examined) meet our requirements and can be used for anonymous PAOs and blind CAs, including popular services that have a large user base: Gmail, Outlook, Hotmail, Mail.com, etc.

5.3 Anonymous PAO for SMTP

Using anonymous PAO, a prover (who owns an email account `alice@domain.com` that is administered by a service `domain.com`) can prove to the verifier that she owns an email account from `domain.com`, without disclosing the exact email address. Unlike conventional PAOs in which a verifier sends a challenge to a prover's account, our anonymous PAO realizations work in the opposite direction. The setup is shown in Figure 5: the prover runs a modified SMTP client, and uses SMTP-STARTTLS to interact with the private SMTP server of `domain.com` via a proxy managed by the verifier using the following protocol:

(1) The verifier (i.e., the proxy in the SCI) determines the address of `smtp.domain.com`, e.g., via DNS, and checks whether the server satisfies the aforementioned requirements.

(2) The prover (i.e., the client in the SCI) sends an email using the SMTP server, to any location only accessible to the prover, via the verifier. Most of the SMTP messages are sent in separate TLS fragments. For the body of the email the client and proxy use a shared template that specifies the location of the challenge in the body and its format (e.g., the challenge should be a certain-length string of random ASCII characters).

(3) The proxy injects a challenge into that email via SCI. The TLS fragment containing the email body is handled via our SCI-TLS injection protocol; this is the Q_u^* message using the notation

from §4. The resulting TLS record will be sent to the server by the proxy, and then the client can finish the session.

(4) The client will later retrieve the email (via an independent and standard connection to its email provider), and extract the challenge.

(5) The client will verify the verifier's identity (e.g, proxy's certificate) and send the challenge to the verifier to complete proof. Assuming only authenticated users can use the server to send emails, this will prove ownership of an account.

Challenge steganography. Recall that one of our security goals is service obliviousness. This is not always important, but could be in some settings. Assuming that SCI-TLS is server oblivious, meaning it is not uniquely identifiable as such, what remains is to ensure that injected messages are not detectable as PAOs. This is fundamentally a task of steganography, but we are aided here by the fact that challenges can be relatively short and the rest of the message can even be hand-crafted.

We designed various example message templates that can be used for hiding a challenge. For example, the message template can be an email that contains a public key or encrypted files (PDF, zip file, etc.); It is easy to embed a short random-string challenge in the template, by simply replacing a portion of the random string with the challenge.

5.4 Blind Certificate Authorities

We now show how to extend our SMTP-based anonymous PAO to build a blind certificate authority service. A blind CA can verify a person's ownership of an email account and then sign an X.509 certificate for the user's public key. The certificate is mainly for binding the public key to the email account and serves as a proof of email account ownership for uses in other privacy-enhancing systems like Cinderella. Unlike conventional CAs, ours will be *blind*: the CA does not learn the user's identity or public key.

At a high level, our blind CA implementation works as follows. (1) The CA runs the SMTP-based anonymous PAO protocol with the client, but additionally extracts from the transcript of the protocol execution a cryptographic commitment to the client's username, `alice`. (2) The CA gives the client a certificate template that is completely filled except for the entries *subject* (the client's username, e.g., `alice`) and *public key*. The client fills in these missing entries, and then sends the hash of this certificate and a zero-knowledge proof to the CA to prove that the same username has been used in the commitment and the certificate. (3) The CA verifies that the proof is correct, signs the hash using a standard digital signature scheme and sends it back. The result is that the client obtains a valid certificate, signed by the CA.

We next elaborate on how to generate a valid commitment during the anonymous PAO, and then describe the certificate generation procedures.

Anonymous PAO with binding identity. For blind CA, we need a slightly stronger form of PAO in which upon acceptance of the statement by the proxy (i.e., being convinced that the client is some eligible user of `domain.com`), it outputs in addition a cryptographic commitment to the identity of the client (henceforth, say, `alice`). The commitment is hiding (and therefore does not reveal the identity of `alice`), but it also *binding* (and so it is infeasible to link this interaction with a different user rather than `alice`).

Ristenpart et al. show that for several ciphersuites for TLS, their ciphertexts bind the underlying plaintexts and can be treated as secure commitments, where the opening is the keys derived from the handshake phase [50]. Such a ciphersuite is AES-CBC with HMAC. In our context of anonymous PAO and SMTP, the encrypted messages that are being delivered by the verifier (i.e., the proxy) to

the server contain the authentication data of the client such as username and password. We refer to this information as the identity of the client, and elaborate how these ciphertexts bind the identity of the client in the sequel. Note that [50] demonstrates an attack that breaks the binding security of AES-GCM, which, however, cannot be directly applied to blind CA because the attacker cannot construct arbitrary messages in our setting — all messages must be semantically meaningful to the SMTP server. Whether AES-GCM can be used as a secure commitment in blind CA remains an open question.

Our SMTP client is instructed to only use the minimum number of commands (`AUTH`, `MAIL`, `RCPT`, and `DATA`) to send an email. Each command and the email content will be sent in one message (i.e., TLS fragment). Thus the first message sent, which is the TLS encryption of the message `AUTH PLAIN [sep]alice[sep]password`, will contain the client's email account, and the fifth message will be the email body into which a challenge will be injected. [5] The procedures are shown in Figure 5. The first message will be taken as the commitment C. The client stores the opening of the commitment (the associated CBC-HMAC keys K_{hmac} and K_{aes}) for late use, The challenge to be injected is a random string M^*, and the proxy adds C to a table under the index M^*.

While running the anonymous PAO (with binding identity), the proxy expects to see exactly five messages (four commands plus an email) sent from the client, and each message is followed by exactly one response from the server (not counting TLS handshake messages and the cleartext `EHLO` at the very beginning). It is important that the proxy aborts after seeing five responses. If it allows more, an attack that abuses the flexibility in number of messages arises; see §5.5.

Some servers might require the `EHLO` after `STARTTLS`. The proxy can check this when examining if the server satisfies the requirements. In this case, the proxy simply needs to let the client use six messages (with six responses) to finish the session, and grabs the second message seen as the commitment.

Certificate generation. The client uses a X.509 certificate template prepared by the CA to generate a legitimate X.509 certificate *cert*, with the *subject* field being set to the client's email account `alice` and the *public key* field being set to the client's public key `pkey`. The other fields (expiration duration, organization, etc.) in the template are public, and their values will be shared with the client and be validated by the CA during certificate generation. The client generates a hash $h = H(cert)$ of its certificate, and produces a zero-knowledge proof, named *CA proof*, that demonstrates (1) her knowledge of the necessary information (the email account and the public key) to form a certificate (the hash of which is h); (2) the knowledge of the underlying message and the opening of the commitment C, namely the secret keys used during the PAO session, and the email account and password in C (which is used to send the challenge M^*); and (3) the account in the *subject* field of *cert* is the same as the account in C. The private witness of the proof consists of the email account, the password, `pkey`, K_{hmac}, and K_{aes}. Note that the proxy will not verify the client's ownership of the public key in the certificate; this can be done by the party to whom the certificate is presented.

Assuming the anonymous PAO is successful, the client can retrieve M^*. It sends M^*, h, as well as the CA proof to the proxy. The proxy retrieves the commitment C based on M^*, and verifies the correctness of the proof. Assuming the zero-knowledge proof is accepted by the proxy, the proxy can be certain that the same email account was used in the anonymous PAO and the certificate over which h is computed. Then, the proxy can sign the hash value h and send the result back to the client.

Anonymous registration. In an anonymous credential system [6, 12, 18, 32, 48], a user can prove to a verifier her ownership of a credential from a CA without revealing the credential. Such systems

[5] `[sep]` is a special character defined in [37]. The account and password are base64 encoded.

aim at providing anonymity and unlinkability, i.e., the verifier cannot learn the identity of the user and multiple uses of the same credential cannot be linked. However, during registration to obtain a credential, existing anonymous credential systems all rely on a trusted third party (the CA) to verify the identity of the user, for example by performing a conventional PAO.

In settings in which users do not want to reveal that they have obtained a credential, we can replace the registration with our blind CA protocol. This allows the user to obtain a credential attesting to ownership of an email address, without revealing to any party that a particular user has obtained the credential.

5.5 Security Analysis

In this section, we discuss several potential application-specific attacks.

Client protocol violation. We start by investigating potential security issues arising from abuse of SMTP semantics. A corrupted client may violate the agreed protocol via extra requests, fragmented commands, or multiple commands per request, potentially violating injection secrecy. For example, the client might send `AUTH PLAIN bob` in the first message, followed by `AUTH PLAIN alice`, to get a certificate for an account `bob` when really only `alice` is owned; or she might split the `AUTH` command into two pieces, and send the second piece and the `MAIL` command in one message. But all will require either more client commands than the number expected or change the request/response sequence. When the target server satisfies the requirement NoPipeline or RFCCompliant, the actual number of client commands sent is visible to the proxy via counting the server's responses. Thus if the proxy detects that the client deviates from the agreed protocol (sending one message but receiving multiple responses, out-of-order requests/responses, etc.), the proxy will immediately terminate the session before the challenge injection.

Proxy injection attacks. The proxy might attempt to violate transcript privacy and client anonymity by injecting a message that contains meaningful SMTP commands. A concrete example is that the proxy can inject a message like "CRLF.CRLF RCPT:... DATA:..." to initiate a new message that will be sent to a proxy-controlled email and learn the email address of the client. This attack can be ruled out if it suffices to restrict the challenge length to at most 19 bytes (152 bits): the mandatory fixed bytes needed (such as CRLFs, command keywords, spaces and newlines) in the commands would be more than 19 bytes. Since the client enforces message length (it computes the hash tag that covers the length), the proxy can't insert anything but the agreed upon amount of bytes. Hence, the attacker will not be able to initiate a new email under this length restriction. Actually, if the target server satisfies the requirement NoPipeline, the attack will not work since a message containing multiple commands will be rejected by the server.

Impersonation and man-in-the-middle attacks. A malicious proxy might announce itself as a verifier proxy and attempt a man-in-the-middle attack: forward back and forth messages between the client and the real verifier's proxy. In this case, the client can still finish a PAO. Transaction privacy guarantees that the malicious proxy cannot learn any messages between the client and the real proxy, but the client may erroneously trust the malicious proxy as if it were the real proxy and later send her (retrieved) challenge to it. A malicious proxy might also perform active man-in-the-middle (MITM) attacks to learn the plaintext messages sent by the client. These attacks are easily prevented by having every proxy set up by the verifier be assigned a certificate, and the client properly verifies the proxy's certificate and the server's certificate before sending her challenge.

6 Implementation and Evaluation

To demonstrate the feasibility of our SMTP-based PAO and blind CA, we focus on AES-CBC with HMAC as a case study. We implemented prototypes of the applications using open-source libraries.

6.1 Implementation

PAO prototype implementation. We use `tlslite` [57] as the TLS library and modify it to add interfaces for extracting the key materials being used in a TLS session, as well as the internal states used in CBC encryption and SHA-256. We use a Python SMTP library `smtplib` to send emails [51]. The client works similarly to a regular SMTP client and follows the SMTP specification. But we do modify the client greeting message to hide client host information.

Malicious secure 2PC. We implement 2-party secure evaluation of AES that is secure against malicious adversaries based on the protocol from Rindal and Rosulek [53] who optimize their implementation in the *offline/online* model. We choose statistical security parameters that offer 2^{-40} security. Two command-line programs are executed by the client and the proxy to compute AES blocks. As in Appendix .3, security is not reduced if the proxy learns the intermediate evaluations of the AES function in the CBC computation. Thus, we implement our overall computation by separately computing each AES block in sequence instead of creating a garbled circuit for the entire computation. The overall computation circuit for one AES block contains less than 32,000 boolean gates, 6,800 of which are AND gates which each require communication between the parties.

Blind CA implementation. We use the ZKBoo framework [27] to create multiple non-interactive zero-knowledge proofs (i.e., *ZKBoo proof*) to construct the CA proof. A CA proof consists of 136 ZKBoo proofs to achieve a soundness error of roughly 2^{-80} as in [27]. Although this framework creates a relatively large CA proof, the operations required to compute the ZKBoo proofs involve only symmetric primitives (unlike other techniques for efficient zero-knowledge which require oblivious transfer). The certificate signature algorithm is SHA-256. We implement ZKBoo versions of SHA-256/HMAC-SHA-256 that support inputs of any length based on the examples provided in [55], and a ZKBoo version of AES-CBC based on the code in [20]. AES S-Boxes are implemented based on [8].

6.2 Evaluation of SMTP-based anonymous PAO

We treat anonymous PAO as a standalone application and measure the latency of the SCI portion across different settings. We hosted the proxy on an m3.xlarge instance in the US-East region of Amazon EC2. The client was running on an Ubuntu 14.04 (64-bit) virtual machine built by VirtualBox 4.3.26, and was configured with 8 GB RAM and 4 vCPUs. We used a tool called `line_profiler` [52] to measure the execution time for each line of code. The sizes of the challenge and message template are fixed as 152 bits and 512 bytes, respectively.

Latency of SCI. We set up our own SMTP server (using Postfix 2.9.6, with pipelining disabled) on the **same** EC2 instance as our proxy to reduce the network latency between the proxy and server, in order to maximize the relative impact of performing SCI. The client ran from two public wireless networks at different locations (labeled as Loc1 and Loc2). And in the best-performing location, we configured the client to use Tor, either with the top 3 high-performance routers or randomly selected routers, to communicate with the proxy. We ran the SMTP anonymous PAO for 50 rounds under each of the settings.

We report on overhead introduced by SCI in Table 6. Tor incurs high overheads as one would expect, so we only report on the best performance. Using public wireless networks achieves better

		Loc1 (No Tor)	Loc2 (No Tor)	With Tor
2P-HMC		0.01 (0.006)	0.03 (0.01)	0.31 (0.15)
2P-CBC	Offline	7.24 (1.65)	8.55 (1.64)	8.10 (3.10)
	Online	0.20 (0.06)	0.35 (0.18)	0.36 (0.16)
Total (**without offline**)		0.76 (0.10)	1.68 (0.11)	4.31 (0.86)
Baseline (SMTP-TLS)		0.31 (0.14)	0.77 (0.45)	3.33 (1.79)

Table 6: The median time and standard derivation (in parentheses) in seconds to complete the 2P-HMAC and 2P-CBC steps, as well as the total PAO and normal SMTP-TLS session durations across 50 executions in each location. "With Tor" show the best-performing setting with Tor being used.

performance in general. The most time-consuming part is offline computation in 2P-CBC; however, it does not rely on inputs and can be even done before establishing the TLS connection. As a baseline, it took the client approximately 0.3 s and 3 to send the same email using conventional SMTP-TLS without and with Tor respectively. Thus the latency overhead of SCI is relatively small.

Tests with real services. We tested our anonymous PAO implementation for SMTP against real services using Loc1 without Tor. The services we chose were Gmail and two SMTP servers at two universities (call them server1 and server2). For each service, we measured 50 times the durations of PAO sessions against normal SMTP-TLS sessions (i.e., the total time spent on issuing a connection, sending an email, and closing the connection). As a baseline, the median duration of normal sessions for Gmail, server1 and server2 were 0.44, 0.93, and 0.79 seconds, while median duration of PAO sessions (without offline stage) were 1.01, 1.64, and 1.53 seconds.

Server obliviousness and session duration. An adversary might attempt to detect SCI by inspecting the SMTP session duration: longer sessions would seemingly be indicative of using SCI. But actually this alone would not be a very good detector. We extracted and analyzed the durations of 8,018 SMTP-STARTTLS sessions from a dataset of terabytes of packet-level traffic traces collected from campus networks.[6] The distribution of the SMTP durations is long-tailed, and about 15% of the SMTP sessions analyzed requiring more than 10 s to complete. This indicates that attempting to detect SCI in such a coarse way will have a high false-positive rate. Of course, there could be more refined detection strategies that take advantage of, for example, inter-packet timing. We leave examining other possible traffic analysis techniques to future work.

6.3 Evaluation of blind CA

We used a 475-byte X.509 certificate template with a 128-bit account/password and 2048-bit public key in our testing. The certificate hash is produced by SHA-256.

CA proof generation and verification time. The size of a ZKBoo proof for a given computation is only decided by the number of AND/ADD gates in the corresponding arithmetic circuit. In our implementation, The total number of AND/ADD gates is 78,064. The resulting ZKBoo proof size was 625,768 bytes. The average computation time for generating one ZKBoo proof was 22.3 ms (over 50 rounds): evaluating one AES block and one round of SHA-256 compression took about 672 us and 586 us on average. Note that we used a byte-oriented, optimized algorithm for AES whereas a naive algorithm for SHA-256, which involves time-consuming copy operations. The verification time for one ZKBoo proof was about 16.3 ms. Based on the design of ZKBoo, the upper bound of the verification time can be approximated by the corresponding proof generation time (in the same

[6] We obtained an IRB exemption to collect the traffic. Here we only used extracted timings from that dataset for our measurements.

setting). The size and computation time of a CA proof increase almost linearly with the number of ZKBoo proofs it has. It took about 2.9 s and 2.3 s to generate and verify a CA proof (136 ZKBoo proofs) respectively. The total size of a CA proof was about 85.1 M. While large, we expect that generating proofs will be an infrequent task in deployments. We also believe that with further code optimization and more advanced ZK techniques, e.g., using Ligero [1], could significantly improve performance.

In Appendix §.1, we estimate the performance impact of using generic MPC techniques for our applications.

7 Related Work

Secure multi–party computation. Secure multi-party computation (MPC) is a technique for multiple parties to jointly compute a function over all their private inputs, while no party can learn anything about the private inputs of the other parties beyond what can be informed from the computation's output [25, 60]. SCI can be seen as a special-case of MPC, but using general-purpose protocols would be more expensive and we limit their use. As such we benefit from the now long literature on making fast MPC implementations [5, 9, 19, 31].

Group and ring signatures. In a group signature scheme [17], a trusted party gives credentials to a group of participants. Any participant can sign a message on behalf of the entire group, with the privacy guarantee that no one aside from the trusted party can learn which member of the group signed the message. This does not provide the participation privacy we seek: the trusted party learns all participants.

Ring signatures [54] do not require a trusted third party, but nevertheless allow signing on behalf of an ad hoc group of public keys. The public keys, however, must be certified by traditional means, meaning ring signatures do not, by themselves, provide the level of participation privacy we seek. That said, ring signatures give a weaker form of participation privacy should there be some deniability in terms of having an ostensible reason for registration of public keys suitable for ring signatures.

Anonymous credentials. Anonymous credentials systems allows a user to prove to another party that she has a valid certificate issued by a given CA, without revealing the content of the certificate [12, 18, 32, 48]. Some systems focus on solving privacy issues that arise during certificate issuance, and allow a user to obtain the signature for a certificate from a CA without revealing privacy-related information in the certificate or, in some cases, without revealing any part of the certificate [6, 49]. However, all these systems rely on a trusted third party that knows the user's identity to perform an initial user registration. Our PAO protocol and blind CA do not.

Multi-context TLS. Multi-context TLS (mcTLS) [44] is a modification to TLS that gives on-path middleboxes the ability to read or write specified *contexts* (i.e., a portion of the plaintext messages) of a TLS session. Each context ends up encrypted under a separate symmetric key, and so some can be shared with the on-path middleboxes. This allows injection of messages, but is not backwards compatible with existing web infrastructure. It also would not be able to achieve service obliviousness, as the server must know which contexts were provided by the proxy.

Mailet. Li and Hopper design a secure computation protocol for TLS GCM and use it to realize a censorship-circumvention system named Mailet [39]. As in our setting, they have a client and proxy that jointly compute a TLS record to allow authentication to a remote server (Twitter in the case of Mailet). While the application setting and goals are different than ours, one might hope to use their protocol to achieve SCI. Unfortunately, to adopt their protocol, the proxy is given the authentication key, allowing it to violate transcript integrity by forging messages. Moreover,

Mailet relies on ad-hoc countermeasure that randomize order and length of application-layer fields, but these won't work for many protocols (including SMTP). Finally, while we note that in theory one can perform anonymous PAOs against authenticated HTTPS services, significant care would be needed to ensure proper understanding of the semantics of the service in order to guarantee security.

8 Conclusion

In this work, we built the first blind CA: an authority that can validate identities and issue certificates without learning the identity. Blind CAs provide a way to construct anonymous credential systems that ensure participation privacy, meaning no single system learns of all the participants. This is important in settings where revealing the users of a system may already put them at risk.

We introduced a number of first-of-their-kind sub-protocols in order to build a blind CA. Secure channel injection (SCI) allows a proxy to inject some (constrained) plaintext content into a stream of encrypted data from a client to server. The client learns nothing about the injected data, while the proxy learns nothing about other messages sent in the stream. We then showed how SCI can be used to perform anonymous proofs of account ownership (PAO) for email accounts. The user can prove ownership of some email address on a service, but not reveal to the verifier which one. Our blind CA protocol checks email ownership via an anonymous PAO, and then uses zero-knowledge proofs to validate and sign an X.509 certificate binding the email to a user's chosen public key, all without ever learning the exact email or public key. Our prototype implementation shows that blind CAs are efficient enough for use in practice, and that they work with existing SMTP services.

Acknowledgments

This work was partially supported by a Junior Fellow award from the Simons Foundation, NSF grants CNS-1330308, CNS-1558500, CNS-1704527, TWC-1646671, TWC-1664445, CNS-1561209, CNS-1217821, CNS-1704788, AFOSR Award FA9550-15-1-0262, a Microsoft Faculty Fellowship, a Google Faculty Research Award, and a gift from Microsoft.

References

[1] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM CCS*, 2017.

[2] Arclab. A list of SMTP and POP3 server. `https://www.arclab.com/en/kb/email/list-of-smtp-and-pop3-servers-mailserver-list.html`, 2016.

[3] M. Belenkiy, J. Camenisch, M. Chase, M. Kohlweiss, A. Lysyanskaya, and H. Shacham. Randomizable proofs and delegatable anonymous credentials. In *CRYPTO*, 2009.

[4] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS*, 1993.

[5] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: A system for secure multi-party computation. In *ACM CCS*, 2008.

[6] V. Benjumea, J. Lopez, J. A. Montenegro, and J. M. Troya. A first approach to provide anonymity in attribute certificates. In *PKC*. 2004.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak and the sha-3 standardization. *NIST*, page 30, 2013.

[8] J. Boyar and R. Peralta. A depth-16 circuit for the AES S-box. *IACR Cryptology ePrint Archive*, 2011:332, 2011.

[9] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. *Network*, 1:101101, 2010.

[10] J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *EUROCRYPT*, 2001.

[11] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, 2004.

[12] J. Camenisch and E. Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM CCS*, 2002.

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

[14] D. Chang and M. Yung. Midgame attacks (and their consequences). `http://crypto.2012.rump.cr.yp.to/008b781ca9928f2c0d20b91f768047fc.pdf`, 2012.

[15] Y. Chang and C. Lu. Oblivious polynomial evaluation and oblivious neural learning. *Theor. Comput. Sci.*, 341(1-3):39–54, 2005.

[16] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, 1985.

[17] D. Chaum and E. Van Heyst. Group signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, 1991.

[18] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno. Cinderella: Turning shabby x.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *IEEE S&P*, 2016.

[19] D. Demmler, T. Schneider, and M. Zohner. Aby-A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[20] dhuertas. AES algorithm implementation using C. `https://github.com/dhuertas/AES`, 2016.

[21] T. Dierks and E. Rescorla". The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, 2006.

[22] T. Dierks and E. Rescorla". The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008.

[23] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman. Neither snow nor rain nor mitm...: An empirical analysis of email delivery security. In *ACM IMC*, 2015.

[24] EEasy. SMTP server list. `http://www.e-eeasy.com/SMTPServerList.aspx`, 2016.

[25] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[26] S. Ghosh, J. B. Nielsen, and T. Nilges. Maliciously secure oblivious linear function evaluation with constant overhead. In *ASIACRYPT*, 2017.

[27] I. Giacomelli, J. Madsen, and C. Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *Usenix Security*, 2016.

[28] O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[29] C. Hazay. Oblivious polynomial evaluation and secure set-intersection from algebraic prfs. In *TCC*, 2015.

[30] C. Hazay and Y. Lindell. Efficient oblivious polynomial evaluation with simulation-based security. *IACR Cryptology ePrint Archive*, 2009:459.

[31] W. Henecka, A.-R. Sadeghi, T. Schneider, I. Wehrenberg, et al. Tasty: Tool for automating secure two-party computations. In *ACM CCS*, 2010.

[32] S. Hohenberger, S. Myers, R. Pass, and A. Shelat. Anonize: A large-scale anonymous survey system. In *IEEE S&P*, 2014.

[33] R. Holz, J. Amann, O. Mehani, M. Wachs, and M. A. Kaafar. TLS in the wild: An internet-wide analysis of tls-based protocols for electronic communication. *NDSS*, 2016.

[34] Internet Security Research Group. Let's encrypt - Free SSL/TLS certificates. `https://letsencrypt.org/`, 2016.

[35] keybase.io. Public key crypto for everyone. `https://keybase.io/`, 2016.

[36] A. Kiayias and M. Yung. Cryptographic hardness based on the decoding of reed-solomon codes. *IEEE Trans. Information Theory*, 54(6):2752–2769, 2008.

[37] J. Klensin. Simple Mail Transfer Protocol. RFC 5321, 2008.

[38] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, 1997.

[39] S. Li and N. Hopper. Mailet: Instant social networking under censorship. volume 2016, pages 175–192.

[40] G. Lindberg. Anti-spam recommendations for SMTP MTAs. RFC 2505, 1999.

[41] D. McGrew and D. Bailey. AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655, 2012.

[42] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. Coniks: Bringing key transparency to end users. In *Usenix Security*, 2015.

[43] M. Naor and B. Pinkas. Oblivious polynomial evaluation. *SIAM J. Comput.*, 35(5), 2006.

[44] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. In *ACM SIGCOMM*, 2015.

[45] Nick Sullivan. Padding oracles and the decline of cbc-mode cipher suites. `https://blog.cloudflare.com/padding-oracles-and-the-decline-of-cbc-mode-ciphersuites/`, 2016.

[46] J. B. Nielsen, T. Schneider, and R. Trifiletti. Constant round maliciously secure 2pc with function-independent preprocessing using LEGO. In *NDSS*, 2017.

[47] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, 2015.

[48] C. Paquin and G. Zaverucha. U-prove cryptographic specification v1.1. Technical report, 2013.

[49] S. Park, H. Park, Y. Won, J. Lee, and S. Kent. Traceable Anonymous Certificate. RFC 5636, 2009.

[50] T. R. Paul Grubbs, Jiahui Lu. Message franking via committing authenticated encryption. In *CRYPTO*, 2016.

[51] B. Peterson. smtplib — SMTP protocol client. `https://docs.python.org/2/library/smtplib.html`, 2016.

[52] rkern. Line profiler and kernprof. `https://github.com/rkern/line_profiler`, 2016.

[53] P. Rindal and M. Rosulek. Faster malicious 2-party secure computation with online/ofine dual execution. In *Cryptology ePrint Archive*, 2016.

[54] R. L. Rivest, A. Shamir, and Y. Tauman. How to leak a secret. In *ASIACRYPT*, 2001.

[55] Sobuno. Zkboo. `https://github.com/Sobuno/ZKBoo`, 2016.

[56] R. Tonicelli, A. C. A. Nascimento, R. Dowsley, J. Müller-Quade, H. Imai, G. Hanaoka, and A. Otsuka. Information-theoretically secure oblivious polynomial evaluation in the commodity-based model. *Int. J. Inf. Sec.*, 14(1):73–84, 2015.

[57] Trevp. Tls lite version 0.4.9. https://github.com/trevp/tlslite, 2016.

[58] X. Wang, S. Ranellucci, and J. Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *ACM CCS*, 2017.

[59] Wikipedia. Open mail relay. https://en.wikipedia.org/wiki/Open_mail_relay, 2016.

[60] A. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science*, 1986.

[61] H. Zhu and F. Bao. Augmented oblivious polynomial evaluation protocol and its applications. In *ESORICS*, 2005.

.1 Using Generic MPC

We give a preliminary analysis of the potential performance impact of using generic MPC to build our applications.

In the anonymous PAOs, using naive implementation to compute the TLS record directly for a message of size 512 bytes would result in a circuit of 0.94 M+ AND gates (6,800 for one AES operation and 90,825 for one SHA-256 operation; 32 AES and 8 SHA-256 operations in total). In contrast, our protocol only performs MPC for 4 AES operations (27,200 AND gates) as opposed to 32 AES operations in generic MPC. That is, if using the same MPC technique, computing a TLS record directly would introduce more than 8x overhead than our protocol. Moreover, our 2P-HMAC protocol does not need MPC at all and only transfers 512-bit of data regardless of the size of the inputs. On the contrary, it already takes the state-of-the-art MPC techniques 10 ms (9 KB data transferred) in the online phase to compute SHA-256 with 256-bit inputs from both parties in the **LAN** setting, not to mention HMAC requires two rounds of SHA-256 and 512-byte input in our application [46, 58].

Recall that during certificate generation one needs to do at least 6 AES and 8 SHA-256 operations for reconstructing the commitment and generating the certificate hash. With naive implementation, such process would result in a circuit of (roughly) 0.95 M+ AND gates. And, using the technique from [58], the total data (including all the phases) that needs to sent from the prover to the verifier will be more than 460 MB, which is much more than the data (i.e., the 85 MB CA proof) sent in our protocol. Even though it might be tolerable for the client as certificate generation is a one-time process, such overhead would cause heavier burden on the CA in practice. Note that our estimations here are simplified, which underestimate the cost of generic MPC. The circuit generated by generic MPC will be more complicated and the actual overhead will be higher on commodity hardware.

.2 TLS with AES-GCM

To broaden the usage of SCI, we implemented SCI atop AES-GCM, which is another widely-supported TLS ciphersuite and gradually gains more users [45].

In order to obtain higher efficiency, we allow the corrupted client to "shift" the message of the proxy. In particular, we modify Functionality 1 and define the following "weaker" SCI functionality, such that it allows a corrupted client to send some message Δ to the functionality (in addition to M_t^p and M_t^s), and the output of the server is $M_t^p, M^* + \Delta, M_t^s$. Moreover, in order to formalize that the input of the proxy is taken from a high-entropy source, we let the trusted party (in case of an honest proxy) to choose the input for it. The functionality is formally described in Functionality 6. In this section, when we relate to secure channel injection, we refer to it with respect to Functionality 6. We adopt Definition 2 similarly, and refer to it with respect to this functionality.

Functionality 6: Weak Message Injection (parameterized with a message template: $M_t = (|M_t^p|, |M^*|, |M_t^s|)$)

- **Input:** The client holds some input prefix message $M_t^p \in \{0,1\}^{|M_t^p|}$ and suffix $M_t^s \in \{0,1\}^{|M_t^s|}$. The server has no input. We assume that the message template is known to the client and the proxy.
  - *Honest proxy (and corrupted client):* The proxy has no input. The functionality chooses $M^* \in \{0,1\}^{|M^*|}$ uniformly at random. The corrupted client inputs some $M_t^p \in \{0,1\}^{|M_t^p|}$, suffix $M_t^s \in \{0,1\}^{|M_t^s|}$ and $\Delta \in \{0,1\}^{|M^*|}$.
  - *Corrupted proxy (and honest client):* The proxy sends some $M^* \in \{0,1\}^{|M^*|}$ to the functionality. The client sends some $M_t^p \in \{0,1\}^{|M_t^p|}$, suffix $M_t^s \in \{0,1\}^{|M_t^s|}$ and the functionality defines $\Delta = 0^{|M^*|}$.
- **Output:** The server outputs $(M_t^p, M^* + \Delta, M_t^s)$. The proxy outputs M^* and the client has no output.

Note that this functionality is weaker than Functionality 1, however, it suffices for our applications. More specifically, as the client is later responsible for "revealing" the challenge M^*, it can extracts it from $M^* + \Delta$ as it knows Δ.

.2.1 2P-CTR and 2P-GMAC

2P-CTR. We now turn to show how to compute a counter mode encryption over $M_t^p||M^*||M_t^s$ given a secret key K. The client holds K, messages $M_t^p = (M_1, \ldots, M_q)$, and $M_t^s = (M_{q+r+1}, \ldots, M_t)$, the proxy holds $M^* = (M_{q+1}, \ldots, M_{q+r})$, where each $M_i \in \{0,1\}^n$.

The counter mode on message $M = (M_1, \ldots, M_t)$ is defined as follows. IV is chosen uniformly at random from $\{0,1\}^{96}$, and we set the counter $J_0 = IV||0^{31}||1$. We define $\mathsf{inc}_s(X)$ to be the function that increments the right-most s bits of the string X, mod 2^s; the left-most $|X|-s$ bits remain unchanged, and define $\mathsf{inc}_s^i(X) = \mathsf{inc}_s(\mathsf{inc}_s^{i-1}(X)$, and $\mathsf{inc}_s(X)^1 = \mathsf{inc}_s(X)$. We define $J_i = \mathsf{inc}_{32}^i(J_0)$ Then, each message $C_i = \mathsf{AES}_K(J_i) \oplus M_i$ for $i = 1, \ldots, t$.

The 2P-CTR protocol is very simple. The client sends the proxy the IV together with the ciphertexts $(C_1, \ldots, C_q)$ and $(C_{q+r+1}, \ldots, C_t)$ corresponding to its messages. Moreover, the client computes also the "key stream" for the proxy, consisting of all keys $\mathsf{AES}_K(J_{q+1}), \ldots, \mathsf{AES}_K(J_{q+r})$. The proxy can then compute the ciphertexts $C_{q+1}, \ldots, C_{q+r}$ using its own messages. The output of the proxy is all cipheretexts $(C_1, \ldots, C_t)$. We denote this protocol as

$$\mathsf{2P\text{-}CTR}((IV, M_t^p, M_t^s), M^*) = (((C_1, \ldots, C_q), (C_{q+r+1}, \ldots, C_t)), (C_{q+1}, \ldots, C_{q+r})) \quad (1)$$

We remark that this protocol allows the client to manipulate the ciphertexts of M^*, and to add to it some Δ. However, as we will see below, this addition can be extracted, and the client knows the "shift" it adds to the proxy's message, which suffices for our applications.

2P-GMAC. For key K, IV, and blocks $X = (X_1, \ldots, X_t)$ where each X_i is of size n bits, we define the function

$$\mathsf{GHASH}_H(X) = Y_{t+1}$$

where Y_{t+1} is defined as follows. For $i = 0$, we define $Y_0 = 0$, and for $i = 1, \ldots, t+1$, we defined Y_i recursively as:

$$Y_i = (Y_{i-1} \oplus X_i) \cdot H \ . \quad (2)$$

Note that in fact, this is equivalent to evaluating the polynomial $p(x)$ on the point H, where the polynomial is defined as $p(x) = \sum_{i=1}^{t} X_i \cdot x^{t-i+1}$. The summation and multiplications are performed in $\mathrm{GF}(2^{128})$.

In our case, we are interested in evaluating this polynomial where the client holds the point H, $X^p = (X_1, \ldots, X_q)$ and $X^s = (X_{q+r+1}, \ldots, X_t)$, and the proxy holds $X^* = (X_{q+1}, \ldots, X_{q+r})$. We can write the polynomial $p(x)$ as a sum of the following three polynomials, representing the different parts (M_t^p, M^*, M_t^s):

(1) $p^p(x) = X_1 \cdot x^t + \ldots + X_q \cdot x^{t-q+1}$.

(2) $p^*(x) = X_{q+1} \cdot x^{t-q} + \ldots + X_{q+r} \cdot x^{t-q-r+1}$.

(3) $p^s(x) = X_{q+r+1} \cdot x^{t-q-r} + \ldots + X_t \cdot x$.

Thus, $p(x) = p^p(x) + p^*(x) + p^s(x)$. The client knows $p^p(\cdot), p^s(\cdot)$ and the point H, whereas the proxy knows only $p^*(\cdot)$. Therefore, we can reduce this computation to oblivious polynomial evaluation. Formally, let

$$F_{\mathsf{ObvPoly}}(H, p^*) := (p^*(H), \lambda) \ ,$$

be the two party functionality in which the client holds a point H, the proxy holds a polynomial $p^*(\cdot)$ and the client receives the evaluation of $p^*(H)$.

Using GHASH, we define $\mathsf{GMAC}(K, IV, X) = \mathsf{GHASH}_H(X) \oplus \mathsf{AES}_K(IV||0^{31}||1)$, where $H = \mathsf{AES}_K(0^{128})$. We describe now the two party protocol that computes GMAC. The protocol 2P-GMAC (Protocol 7) is described in the F_{ObvPoly}-hybrid model, and we discuss how to implement the F_{ObvPoly}-functionality after. In the protocol, the proxy we will add a random constant term to the polynomial $p^*(\cdot)$, in order to mask the result $p^*(H)$.

Protocol 7: 2P-GMAC $((K, IV, X^p, X^s), Y^*)$ in the F_{ObvPoly}-hybrid model
Input: The client holds K, IV, and the blocks $X^p = (X_1, \ldots, X_q)$, and $X^s = (X_{q+r+1}, \ldots, X_t)$; the proxy holds $X^* = (X_{q+1}, \ldots, X_{q+r})$, where each $X_i \in \{0,1\}^n$.
The protocol:

(1) The proxy defines the polynomial $p^*(x) = \sum_{j=1}^{r} X_{q+j} \cdot x^{r-j+1}$. It then chooses a random field element α and defines $p'(x) = p^*(x) + \alpha$.

(2) The two parties invoke the functionality $F_{\mathsf{ObvPoly}}(H, p') = (p'(H), \lambda)$ where the client inputs the point $H = \mathsf{AES}_K(0^{128})$, and the proxy inputs the polynomial $p'(x)$. The client receives the point $p'(H) = p^*(H) + \alpha$.

(3) Using the values $H, X_1, \ldots, X_q$, the client computes $p^p(H)$. Using the values $H, X_{q+r+1}, \ldots, X_t$, the client computes $p^s(H)$.

(4) The client sends the proxy the tag $T' = p^p(H) + p'(H) + p^s(H) + \mathsf{AES}_K(IV||0^{31}||1) = p(H) + \alpha + \mathsf{AES}_K(IV||0^{31}||1)$. The proxy removes the mask α and obtain the tag $T = p(H) + \mathsf{AES}_K(IV||0^{31}||1)$.

Output: The proxy outputs T.

Implementing F_{ObvPoly}. In order to implement the oblivious polynomial evaluation protocol, we use the protocol of Ghosh, Nielsen and Nilges [26]. This protocol is secure in the malicious setting assuming the existence of an oblivious transfer and noisy encodings, and requires $O(t)$ OTs (or, exponentiations), where t is the degree of the polynomial (in our case, the number of blocks). In anonymous PAOs, the number of OTs is at most 32, which can be done efficiently. We refer also to [29,30,43] for additional oblivious polynomial evaluation protocols based on other assumptions in the malicious setting, and for [15,36,56,61] for protocols in the semi-honest settings. As opposed to

our TLS with CBC and HMAC in which the proxy must inject at least two blocks, here there is no such restriction, and we can allow injection of a single block. In that case, the functionality F_{ObvPoly} is in fact oblivious linear evaluation (OLE), that can be realized using highly efficient protocols (see [26]).

The SCI Protocol. We are now in position to describe our solution for SCI with TLS where the proxy wants to inject a message at some designated point into the stream of encrypted client-to-server message data. Let $Q_1, \ldots, Q_u^*, \ldots, Q_v$ be the sequence of TLS plaintext fragments sent from the client to the server in separate record layer encryptions, with Q_u^* representing the fragment within which the proxy will inject its private message M^*. For simplicity and ease of exposition, we assume that the blocks of the proxy and the client are multiplicatives of the block size, 128 bits.

The encryption of the AES-GCM works as follows. Let K be the key that the client and server shared. Let $M_t^p = (M_1, \ldots, M_\ell)$ be the messages of the client, $M_t^s = (M_{\ell+k+1}, \ldots, M_m)$, and let $M^* = (M_{\ell+1}, \ldots, M_{\ell+k})$ be the messages of the proxy. The secure channel injection protocol is as follows:

(1) The client chooses a random IV of 96 bits.

(2) Call $\mathsf{2P\text{-}CTR}$ where the client inputs IV and M_t^p, M_t^2, the key K, while the proxy inputs M^*. Let $(C_1, \ldots, C_q)$ and $(C_{q+r+1}, \ldots, C_t)$ be the output of the client, and let $(C_{q+1}, \ldots, C_{q+r})$ be the output of the proxy.

(3) Let A be the data that is being authenticated but not encrypted, where $|A| = u$ is of size ≤ 128 ($u = 104$ in TLS 1.2). Let $A' = A||0^{128-u}$. The parties engage in $\mathsf{2P\text{-}GMAC}$ where the client inputs IV, K, A', $(C_1, \ldots, C_q)$ and $(C_{q+r+1}, \ldots, C_t, \mathsf{len}(A)||\mathsf{len}(C))$ and the proxy inputs $(C_{q+1}, \ldots, C_{q+r})$. The proxy receives as output a tag T.

(4) The client sends the proxy the value IV and all ciphertexts. The proxy sends to the server $IV, (C_1, \ldots, C_t)$ and the tag T.

We discuss the security of this protocol in Appendix .4.

Theorem 8. *Modeling* AES *as an ideal cipher, the SCI-TLS protocol is a secure channel injection protocol for TLS with AES-GCM mode (i.e., satisfies Definition 2 with respect to Functionality 6).*

.3 Security Proofs for our SCI-TLS Protocols

We follow the standard definition of secure computation protocols in the presence of a malicious adversary in the stand alone settings, as well as hybrid models, and refer to [13, 28] to the formal definition.

Modeling f and AES. In the case of CBC with HMAC, We model f as a random oracle and AES as an ideal cipher. In the case of AES-GCM, there is no function f, but we still model AES as ideal cipher. We prove the security of the protocols in which all parties have access to the same oracles $f : \{0,1\}^v \times \{0,1\}^d \to \{0,1\}^v$, $\mathsf{AES} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^n$ and $\mathsf{AES}^{-1} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^n$, where for every key $K \in \{0,1\}^d$ and every message $M \in \{0,1\}^n$ it holds that $\mathsf{AES}(K, \mathsf{AES}^{-1}(K, M)) = \mathsf{AES}^{-1}(K, \mathsf{AES}^{-1}(K, M)) = M$. The functions $f, \mathsf{AES}, \mathsf{AES}^{-1}$ are chosen uniformly at random in the respective domain in the initial phase of the execution. We assume that the distinguisher (between the ideal and the real) does not receive access to the oracles, and its knowledge about the oracles is limited to the queries that the adversary $\mathcal{A}$ made during the execution.

.3.1 Security of SCI-TLS with CBC and HMAC

We are now ready to the security proof of the SCI-TLS protocol. We first prove the protocol in the F_{AES}-hybrid model and in the aforementioned random oracle (or ideal cipher model). In fact, we will show that the protocol is statistically-secure in this hybrid model. We then replace the f-random oracle with SHA256 and AES with an AES implementation, and derive security in F_{AES} heuristically according to the random oracle methodology [4]. We then derive security in the plain model by replacing the F_{AES} functionality with the instantiated AES function, with a general-purpose secure computation protocol computing this functionality. Security is concluded due to sequential composition [13, 28]. We prove:

Theorem 9 (Theorem 5, restated)**.** *The SCI-TLS protocol is a secure channel injection protocol for TLS with* $\mathsf{AES\text{-}CBC}$ *and HMAC-SHA-256 (i.e., satisfies Definition 2), assuming that f is a random oracle and* AES *is an ideal cipher.*

We prove the Theorem with respect to the adjusted injection functionality (i.e., Functionality 6), but in which the trusted party always sets $\Delta = 0$, i.e., the client cannot "shift" the output of the proxy.

Proof. For simplicity, we first focus the case where the message template is such that $|M^*| \geq 2n$. Moreover, for simplicity we ignore for now the SQN and HDR and alignment of the messages of the MAC and the AES, and just assume that all messages $|M_t^p|$, $|M_t^s|$ and $|M^*|$ are multiples of d and n. We will later show how to adjust the proof for the other cases as well. We prove separately the case where the client is corrupted and where the proxy is corrupted.

Corrupted client. This case is essentially the "injection secrecy", that is, the client cannot learn M^* during the protocol interaction. Formally, we show that this is the case by constructing a simulator that does not know M^*, but is able to simulate the view of the client, which consists of the messages computed by the proxy and the server, and extracting its input.

For simplicity, we describe the simulator $\mathcal{S}$ as it receives an access to some random instance of the random oracle $f, \mathsf{AES}, \mathsf{AES}^{-1}$, and note that the simulator could have simulated these queries on its own.

The simulator $\mathcal{S}$. The simulator $\mathcal{S}$ works as follows:

(1) *It invokes the adversary $\mathcal{A}$ on an auxiliary input z. Throughout the execution, $\mathcal{S}$ answers the queries of $\mathcal{A}$ using its own oracles $f, \mathsf{AES}, \mathsf{AES}^{-1}$, and it just forwards these queries.*

(2) *$\mathcal{S}$ simulates the TLS handshake protocol as an honest server, and thus learns the keys K_{aes}, K_{hmac}.*

(3) *The simulator chooses a message M_0^* uniformly at random. It simulated an honest proxy interacting with the client in 2P-HMAC and 2P-CBC with the input M_0^*, and also simulates the F_{AES}-functionality inside 2P-CBC (where it just receives some key from the adversary).*

(4) *Using the ciphertexts $C_0, \ldots, C_t$ and the key K_{aes}, the simulator extracts the underlying plaintexts, M_t^p, M_t^s, and the tag T. Then, it verifies the MAC. If the tag is valid, it sends the messages M_t^p, M_t^s as the input of the corrupted client. Otherwise, it sends $\perp$.*

The only difference between the real and the ideal executions is the message M^* that the simulator uses. In particular, in the ideal execution the simulator does not know the message M^* that the proxy receives from the trusted party, and it simulates the protocol with respect to a different random message M_0^* for the proxy. As a result, the view of the adversary in the ideal

corresponds to M_0^*, while the inputs/outputs of the honest parties correspond to $M^* \neq M_0^*$. In the real execution, both the view and the inputs/outputs of the honest parties correspond to M^*.

We show that the unless $\mathcal{A}$ makes some specific queries the random oracle, its view is in fact independent to the message M^* and therefore the distinguisher cannot distinguish between the two executions. Moreover, we show that the probability that a poly-query adversary (even with unbounded computational power) makes these particular queries is exponentially small, for a polynomial poly, and therefore the distinguishing probability of the distinguisher is exponentially small.

Towards this end, let $M^* = (M^*_{\ell+1}, \ldots, M^*_{\ell+j}) = (P^*_{q+1}, \ldots, P^*_{q+r})$ be the injected message (for both the ideal and real). We define the set of *offending queries* in the real execution, denotes by S_{real}, to be one of the following queries:

- ***f*-queries:** Any one of the queries that an honest proxy / simulator make in order to compute $s_{\ell+j+1}$ (given s_ℓ that was sent by the adversary). That is, $s_{\ell+1} = f(s_\ell, M^*_{\ell+1}), \ldots, s_{\ell+j+1} = f(s_{\ell+j}, M^*_{\ell+j})$.
- **AES-queries.** Any one of the AES-queries that an honest proxy / simulator make in order to compute $C_{q+1}, \ldots, C_{q+r}$ by the F_{AES} invocations, given the value C_q sent by the adversary. That is, $C_{q+1} = \mathsf{AES}(K_{aes}, C_q \oplus P^*_{q+1}), \ldots, C_{q+r} = \mathsf{AES}(K_{aes}, C_{q+r-1} \oplus P^*_{q+r})$.
- **AES^{-1}-queries.** Any one of the queries that are correlated to $C_{q+1}, \ldots, C_{q+r-1}$ (note that the adversary can make a query to $\mathsf{AES}^{-1}(K_{aes}, C_{q+r})$, as its view contains C_{q+r}).

Let S_{ideal} denote the set of corresponding queries in the ideal execution, defined with respect to M_0^*. Let B denote the (bad) even in which $\mathcal{A}$ makes a call to any one of the queries in S_{real} and S_{ideal} in the corresponding execution. Since M^*, M_0^* are distributed uniformly, each query contains a block value that is hidden from the view of the adversary unless the adversary makes this specific query. With each query, the adversary $\mathcal{A}$ learns one value, which is also distributed uniformly. As such, it is easy to see that as long as no offending query is made, the view of the adversary is distributed identically in both executions as the view is completely independent to M^*, M_0^*. Moreover, the event B has the same probability in both processes. This implies that $\Pr[B] \leq (|S_{\text{ideal}}| + |S_{\text{real}}|) \cdot (q+1) \cdot (2^{-n} + 2^{-d})$, where q bounds the number of queries that $\mathcal{A}$ makes, n is the block size of AES and d is the size of the message part in HMAC.

Corrupted proxy. In this case, we essentially prove the "transport privacy", that is, that the proxy does not learn anything about other messages other than M^*. Moreover, we show "transcript integrity", and that the proxy cannot modify parts of the message transcript besides M^*. We proceed with a description of the simulator $\mathcal{S}$:

(1) *The simulator plays the role of the client and the server in the handshake phase, and transmits the message throw $\mathcal{A}$. It learns K_{aes} and K_{hmac}.*

(2) *The simulator chooses arbitrary inputs M_t^p, M_t^s of the corresponding sizes, and runs an honest execution of the protocol with respect to this messages.*

(3) *Let $C'_1, \ldots, C'_q$ be the messages sent by the adversary to the simulator, that are supposed to be delivered to the server (these might be different than the ciphertexts replied by the simulator during 2P-CBC). $\mathcal{S}$ decrypts $C'_1, \ldots, C'_q$ using K_{aes} and learns the underlying plaintext $P'_1, \ldots, P'_q$. It acts like an honest server and checks that they messages are well-formed according to the TLS protocol. It extracts the underlying messages $\tilde{M}_t^p \| \tilde{M}^* \| \tilde{M}_t^s$, and verifies the tag.*

(4) *It checks that the extracted messages are those used in the simulation, that is,* $\tilde{M}_t^p = M_t^p$ *and* $\tilde{M}_t^s = M_t^s$.

(5) *If all verifications pass, the simulator sends* $\tilde{M}^*$ *to the trusted party. Otherwise, it sends* $\perp$.

We prove that the real and ideal are indistinguishable via a sequence of hybrids:

- $\mathsf{Hyb_0}$: This is the ideal execution.
- $\mathsf{Hyb_1}$: In this execution, the simulator receives from the trusted party the true inputs M_t^p and M_t^s that the honest client uses in the ideal execution. The simulator uses these inputs instead of choosing arbitrary inputs in Step 2.
- $\mathsf{Hyb_2}$: This is like Hyb_1, while the simulator omits the check whether $\widetilde{M_t^p} = M_t^p$ and $\widetilde{M_t^s} = M_t^s$ in Step 4, and just always sends $\widetilde{M^*}$ to the trusted party in case where the simulated server does not output $\perp$.
- $\mathsf{Hyb_3}$: This is the real execution.

The proof proceeds by showing that Hyb_0 is statistically-close to Hyb_1 due to the random oracle. Specifically, this follows the same argument as the case of a corrupted client, where here the offending queries are some f-queries, $\mathsf{AES}_K(\cdot)$ and $\mathsf{AES}_K^{-1}(\cdot)$-queries:

- ***f*-queries:** Let $s_0 = f(IV, K_{hmac} \oplus ipad)$, $s_1 = f(s_0, M_1), \ldots, s_\ell = f(s_{\ell-1}, M_\ell)$.
- **AES-queries:**
  - Queries related to M_t^p: $C_1 = \mathsf{AES}_{K_{aes}}(IV \oplus P_1), \ldots, C_q = \mathsf{AES}_{K_{aes}}(C_{q-1} \oplus P_q)$.
  - Queries related to M_t^s: $C_{q+r+1} = \mathsf{AES}_{K_{aes}}(C_{q+r} \oplus P_{q+r+1}), \ldots, C_t = \mathsf{AES}_{K_{aes}}(C_{t-1} \oplus P_t)$
- The corresponding queries to AES^{-1}.

We remark that we must set $r \geq 2$. Security is obtained since both K_{hmac} and K_{aes} are distributed uniformly and are hidden from the view of the adversary, as long as no offending query is performed. in the real execution. We define the set of *offending queries* S_{real} to be the set of the following queries:

Hyb_1 and Hyb_2 are also statistically-close due to the random oracle. Fix the oracle AES and AES^{-1}. This fixes the messages $\widetilde{M_t^p}$ and $\widetilde{M_t^s}$ (that are differ than M_t^p and M_t^s) and the message $\widetilde{M^*}$. We again define a set of offending queries to the oracle f (the set of queries that are necessary for computing the tag T' for the message $(\widetilde{M_t^p}, \widetilde{M^*}, \widetilde{M_t^s})$). First, observe that T' is an output of f, and therefore, except for some exponentially-small probability, the adversary must receive T' as an output of an offending f-query. Statistically-closeness is then obtained since k_{hmac} is chosen uniformly at random and f is a random oracle, all offending queries are hard to guess.

This concludes the proof of Theorem 9. ∎

Dealing with block alignments. Due to alignment issues and the messages SQN, HDR, the client has to share parts of its message with the proxy. We first deal with case (a) in Figure 4. We parametrize Functionality 1 with the leakage function

$$\mathcal{L}((M_t^{p_1}, M_t^{p_2}), (M_t^{s_1}, M_t^{s_2}) = (M_t^{p_1}, M_t^{s_1})$$

where the sizes are defined as in Figure 4. Then, we essentially refer to the message $\hat{M}^* = (M_t^{p_1}, M^*, M_t^{s_1})$ as the injected message in the 2P-HMAC and the message M^* in 2P-CBC. As

in the current proof, the simulator decrypts the messages that the corrupted proxy transmits at the last stage of the protocol to the server, recovers the underlying plaintexts and the tag T, and verifies that the tag is valid and correspond to the M_t^p, M_t^s. If there is a difference between the ideal and the real, we can use this adversary to forge an HMAC, exactly as the previous reduction.

Dealing with cases where $|M^*| \leq d$. We mentioned in Section 4.1 the special case in which the proxy wants to inject only one AES-block. In this case we force $|M_t^s| = 0$ and the client sends the HMAC tag to the proxy. We have to show that this tag can be simulated as well, in the case of a corrupted proxy.

In particular, after the client receives $s_{\ell+k}$ from the corrupted proxy (note that $|M_t^s| = 0$ and therefore $s_{\ell+k} = s^*$, see Protocol 3), then the client computes $T = H(K_{hmac} \oplus opad) \| s^*)$, but this time it sends the tag T to the proxy.

In the ideal execution, the adversary sends $s_{\ell+k}$ to the simulator, which continues to query the random oracle in order to compute $H(K_{hmac} \oplus opad) \| s^*)$ just as in the real execution.

In order to show indistinguishability between the real and ideal, we let the set of offending queries to contain the additional queries to f during the computation of $H(K_{hmac} \oplus opad) \| s^*)$. We claim again that the adversary cannot make these queries as it has no information regarding K_{hmac}. The reduction to HMAC in case the adversary replaces M_t^p or M_t^s follows as well, as the tag T that the simulator (or the adversary $\mathcal{A}_{HMAC}$) computes is indistinguishable from the real tag.

.4 Security Analysis for SCI-TLS with AES-GCM

Theorem 10 (Theorem 8, restated)**.** *The SCI-TLS protocol is a secure channel injection protocol for TLS with AES-GCM mode (i.e., satisfies Definition 2 with respect to Functionality 6).*

Proof. We distinguish between the cases of corrupted client and corrupted proxy.

Corrupted client. The case is essentially the "injection secrecy". We show that the client cannot learn M^*. Towards that end, we show a simulator that does not receives M^* as input, and successfully simulates the view of a corrupted client in the protocol execution. The corrupted client receives an oracle access to $\mathsf{AES}, \mathsf{AES}^{-1}$, and the proof (for the case of a corrupted client) works for any (valid) instantiation of these oracles.

The simulator $\mathcal{S}$. On auxiliary input z the simulator $\mathcal{S}$ works as follows:

(1) *It invokes the adversary $\mathcal{A}$ on an auxiliary input z. Throughout the execution, $\mathcal{S}$ answers the queries of $\mathcal{A}$ using its own oracle $\mathsf{AES}, \mathsf{AES}^{-1}$, and it just forwards these queries.*

(2) *$\mathcal{S}$ simulates the TLS handshake protocol as an honest server with the adversary, and thus learns the key K.*

(3) *$\mathcal{S}$ runs an honest execution of the protocol with the corrupted client $\mathcal{A}$ and input $M^* = 0$ for the simulated proxy. During this execution, it also simulates the execution of F_{ObvPoly}. Let $(K_{q+1}, \ldots, K_{q+r})$ be the key stream the client sent the proxy (each K_{q+i} is supposed to be $\mathsf{AES}_K(ctr_{32}^i(IV))$ for $i = 1, \ldots, r$), and let α_0 be the constant term that the simulator chooses (for the simulated proxy). Let $\widetilde{H}$ be the value the adversary sends to F_{ObvPoly} and let β be the computed result, i.e., $\beta = K_{q+1}\widetilde{H}^{t-q} + \ldots + K_{q+r}\widetilde{H}^{t-q-r+1} + \alpha_0$.*

(4) *If the output of the simulated server is $\perp$, then the simulator sends $\perp$ to the trusted party. Otherwise, let the output of the simulated server be (M_t^p, Δ, M_t^s). The simulator checks in addition that $\widetilde{H} = \mathsf{AES}_K(0^{128})$, and if that holds, then it sends (M_t^p, Δ, M_t^s) as the input of the corrupted client.*

(5) *$\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.*

Claim 11. *For every (unbounded) adversary $\mathcal{A}$ making polynomial number of queries to the random oracle, it holds that:*

$$\{\mathtt{IDEAL}_{F_{MI},\mathcal{S}}(\vec{x},z)\}_{\vec{x},z} \equiv \{\mathtt{REAL}_{\Pi,\mathcal{A}}(\vec{x},z)\}_{\vec{x},z}$$

Proof. Note that there are two differences between the real and ideal executions: In the ideal execution, the simulator simulates the view of the adversary with respect to $M^* = 0$, whereas the output of the execution is with respect to some value M^* that the trusted party chose. In the real execution, both the view of the adversary and the output of the execution are with respect to the same message M^*. In addition, the simulator verifies that the value $\widetilde{H}$ that the adversary uses in the functionality F_{ObvPoly} satisfies $\widetilde{H} = \mathsf{AES}_K(0^{128})$, whereas in the real execution there is no such verification. We prove the claim using a sequence of hybrids:

- **Hyb_0:** This is the ideal execution.
- **Hyb_1:** Here, in Step 3, instead of choosing α_0 uniformly at random as the constant term of the polynomial in the simulator F_{ObvPoly}, the simulator chooses β at random and sets: $\alpha_0 = \beta - K_{q+1}\widetilde{H}^{t-q} - \ldots - K_{q+r}\widetilde{H}^{t-q-r+1}$. When the simulator receives the tag T from the adversary, it subtract α_0 and performs the verification as in the real execution, i.e.,

$$\begin{aligned} T - \alpha_0 = \mathsf{AES}_K(IV||0^{31}||1) + p^p(H) + p^s(H) \\ + K_{q+1}H^{t-q} + \ldots + K_{q+r}H^{t-q-r+1} \ . \end{aligned} \tag{3}$$

  The output of the execution is the output of all parties, as in the ideal execution.

- **Hyb_2:** This is as Hyb_1 with the following modifications. First, the simulator receives from the trusted party the message $M^* = (M_1^*, \ldots, M_r^*)$ that was chosen by the trusted party for the proxy. It uses $((K_{q+1}+M_1^*), \ldots, (K_{q+r}+M_r^*))$ as the ciphertexts of its messages, and uses these values as the inputs to F_{ObvPoly}, with $\alpha_{M^*} = \beta - (K_{q+1}+M_1^*)\widetilde{H}^{t-q} - \ldots - (K_{q+r}+M_r^*)\widetilde{H}^{t-q-r+1}$. The check that the simulated server performs is

$$\begin{aligned} T - \alpha_{M^*} = \mathsf{AES}_K(IV||0^{31}||1) + p^p(H) + p^s(H) \\ + (K_{q+1} + M_1^*)H^{t-q} + \ldots + (K_{q+r} + M_r^*)H^{t-q-r+1} \ . \end{aligned} \tag{4}$$

  In case where the verification holds, let $(M_t^p, \widetilde{M^*}, M_t^s)$ be the output of the simulated server. The simulator sets $\Delta = \widetilde{M^*} - M^*$, and sends the trusted party (M_t^p, Δ, M_t^s).

- **Hyb_3:** Here, in addition, the simulator does not make the check $\widetilde{H} = \mathsf{AES}_K(0^{128})$. It just checks that the tag is correct, and then it sends (M_t^p, Δ, M_t^s) to the trusted party.
- **Hyb_4:** Here, the output of the execution is the output of the adversary, and the output of the simulated proxy and simulated server (instead of the output of the trusted party). This is equivalent to the real execution.

Hyb_0 and Hyb_1. It is easy to see that Hyb_0 and Hyb_1 are identical as there is 1-to-1 transformation between α_0 and β.

Hyb_1 and Hyb_2. We now show that the outputs of Hyb_1 and Hyb_2 are also identically-distributed. First, we observe that the view of the adversary is the same. Therefore, the distributions of all the

values it outputs are the same. We now have two cases, depending on whether $\widetilde{H} = \mathsf{AES}_K(0^{128})$ or not. Clearly, if $\widetilde{H} \neq \mathsf{AES}_K(0^{128})$ then in both cases the output the two executions is $\perp$. Otherwise, we also claim that both executions have the exact same output. Specifically, in Hyb_1, the simulator performs the check whether Eq. (3) holds, and in case it does, the simulator decrypts the ciphertexts and sends the trusted party (M^*, Δ, M_t^s). The output of the server in this execution would be $(M_t^p, M^* + \Delta, M_t^s)$, where M^* is the value chosen by the trusted party. In case of Hyb_2, if Eq. (4) holds then the simulator decrypts the ciphertexts (which are decrypted to $(M_t^p, M^* + \Delta, M_t^s)$), and then subtract M^* from the injected message. It sends the trusted party (M_t^p, Δ, M_t^s) as in Hyb_1, and the output of the server in this execution would be again $(M_t^p, M^* + \Delta, M_t^s)$. In order to conclude this part of the proof, we claim that Eq. (3) holds if and only if Eq. (4) holds. This is true as long as $\widetilde{H} = H = \mathsf{AES}_K(0^{128})$.

Hyb_2 and Hyb_3. We proceed to show that Hyb_2 and Hyb_3 are statistically-close. There are few cases to consider. First, if $\widetilde{H} = H = \mathsf{AES}_K(0^{128})$, then the output of both executions is the same, and depends on whether Eq. (4) holds or not. The second case is when $\widetilde{H} \neq H \ \ (= \mathsf{AES}_K(0^{128}))$. Here, we have two sub-cases to consider. In Hyb_2, when $\widetilde{H} \neq H \ \ (= \mathsf{AES}_K(0^{128}))$ occurs the simulator sends $\perp$ to the trusted party. In Hyb_3, we check whether Eq. (4) holds, where if it holds the simulated sends the decrypted inputs to the trusted party, and in case it does not hold, it sends $\perp$. Therefore, we have a difference between the two execution in case Eq. (4) holds.

We now show that in case $\widetilde{H} \neq H$ then Eq. (4) holds with small probability. Specifically, the adversary already "committed" to $\widetilde{H}$, as long as the tag T, and it has no information about M^*, which is chosen uniformly at random by the trusted party and is given to the simulator. Eq. (4) holds only when α_{M^*} is "correct", i.e., when

$$\begin{aligned}&(K_{q+1} + M_1^*) \cdot \widetilde{H}^{t-q} + \ldots + (K_{q+r} + M_r^*) \cdot \widetilde{H}^{t-q-r+1} \\ &= (K_{q+1} + M_1^*) \cdot H^{t-q} + \ldots + (K_{q+r} + M_r^*) \cdot H^{t-q-r+1}\end{aligned}$$

Fix $(M_2^*, \ldots, M_r^*)$. The above holds if and only if

$$M_1^*(\widetilde{H}^{t-q} - H^{t-q}) = \gamma$$

for some constant $\gamma = K_{q+1} \cdot (H^{t-q} - \widetilde{H}^{t-q}) + (K_{q+2} + M_2^*) \cdot (H^{t-q-1} - \widetilde{H}^{t-q-1}) + \ldots + (K_{q+r} + M_r^*) \cdot (H^{t-q-r+1} - \widetilde{H}^{t-q-r+1})$. As M_1^* is chosen uniformly at random in $GF(2^{128})$, the probability that this verifications holds is exponentially small.

It is easy to see that the outputs of Hyb_3 and Hyb_4 are identical. ■

Corrupted proxy. In this case, we essentially prove the "transport privacy", that is, that the proxy does not lean anything about other message rather than M^*. Moreover, we also show integrity, and that the proxy cannot modify parts of the message transcript besides M^*. We proceed with a formal description of the simulation $\mathcal{S}$.

The simulator $\mathcal{S}$. *On auxiliary input z the simulator $\mathcal{S}$ works as follows:*

(1) *The simulator invokes the adversary $\mathcal{A}$ on an auxiliary input z.*

(2) *$\mathcal{S}$ runs an honest execution between the client and the server of the TLS handshake protocol to simulate the view of the proxy in that protocol. Specifically, it sends each message from the client (resp. server) to the adversary and expects to see that the adversary delivers the message to server (resp. client). As the simulator plays the role of the client and the server it learns the key K, while the adversary does not.*

(3) *$\mathcal{S}$ chooses M_t^p and M_t^s arbitrarily of the corresponding sizes.*

(4) *$\mathcal{S}$ runs an honest execution of the client interacting with the adversary (as a proxy), in 2P-CTR and 2P-GMAC, using the key K and the messages M_t^p, M_t^s and $\Delta = (0^\ell)^r$.*

(5) *At the end of the execution, the simulated server outputs either $\perp$ or some $(\widetilde{M_t^p}, \widetilde{M^*}, \widetilde{M_t^s})$. In the former case, $\mathcal{S}$ sends $\perp$ to the trusted. In the latter case, the simulator first verifies that $\widetilde{M_t^p} = M_t^p$ and $\widetilde{M_t^s} = M_t^s$. If they are equal, it sends $\widetilde{M^*}$ to the trusted party as the input of the corrupted proxy. Otherwise, it again sends $\perp$ to the trusted party.*

(6) *The simulator $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.*

Claim 12. *Assuming that the TLS handshake is a secure key-agreement protocol in the presence of an eavesdropper adversary, for every adversary $\mathcal{A}$ it holds that:*

$$\{\texttt{IDEAL}_{F_{MI},\mathcal{S}}(\vec{x},z)\}_{\vec{x},z} \overset{s}{\approx} \{\texttt{REAL}_{\Pi,\mathcal{A}}(\vec{x},z)\}_{\vec{x},z}$$

Proof. We again prove the claim using sequence of hybrid experiments.

- **Hyb_0**: This is the ideal execution.
- **Hyb_1**: In this execution, the simulator receives from the trusted party the true inputs M_t^p and M_t^s that the honest client uses in the ideal execution.
- **Hyb_2**: This is like Hyb_1, while the simulator omits the check whether $\widetilde{M_t^p} = M_t^p$ and $\widetilde{M_t^s} = M_t^s$, and just always sends $\widetilde{M^*}$ to the trusted party in case where the simulated server does not output $\perp$.
- **Hyb_3**: This is the real execution.

Hyb_0 and Hyb_1: We first claim that Hyb_0 and Hyb_1 are statistically-close. More specifically, any distinguisher cannot distinguish between Hyb_0 and Hyb_1 unless the adversary $\mathcal{A}$ makes some specific queries to the random oracle (similarly to Theorem 9).

The view of the corrupted proxy consists of IV, all ciphertexts $(C_1,\ldots,C_q),(C_{q+r+1},\ldots,C_t)$, the key stream and the tag T'. Each one of these are ciphertexts – are XORed with $\mathsf{AES}_K(ctr_{32}^i(IV||0^{31}||1))$ for some $i = 1,\ldots,t$ (observe that even the tag T' is XORed with $\mathsf{AES}_K(IV||0^{31}||1)$). Thus, unless the adversary makes some specific $\mathsf{AES}_K(\cdot)$-queries (and corresponding AES_K^{-1})-used to encrypt – it cannot distinguish the difference between Hyb_0 and Hyb_1. As the key K is hidden from the view of the adversary, it must guess K in order to make these queries (recall that the random oracle is $\mathsf{AES} : \{0,1\}^d \times \{0,1\}^n \to \{0,1\}^n$ – and the adversary has to specify K when querying the oracle). Let B dente the (bad) event in which makes a call to anyone of these queries. The probably of the event B is bounded by $2\cdot(t+1)\cdot(q+1)\cdot 2^{-|K|}$, where q bounds the number of queries $\mathcal{A}$ makes.

Hyb_1 and Hyb_2: Forging the MAC. Next, we claim that no distinguisher Hyb_1 and Hyb_2 with non-negligible probability. In fact, we claim that the event in which the simulated server outputs values which are differ than $\perp$, but then the simulator sends $\perp$ to the trusted party occurs with only negligible probability.

In particular, assuming that the above does not hold and that there exists an adversary $\mathcal{A}$ that is interacting with $\mathcal{S}$ (with some messages M_t^p and M_t^s), and at the end of the execution, the simulated server outputs a message that is differ than M_t^p or M_t^s. Then, we construct an adversary $\mathcal{A}_{MAC}$ that receives a MAC oracle and succeeds to forge a MAC on a message that it did not query before as follows:

(1) It invokes the adversary $\mathcal{A}$ on the auxiliary input z.

(2) It receives an oracle access to the encryption oracle on some key K chosen by the challenger. It sets $M^* = (0^d)^r$, and queries the encryption oracle on message $M_t^p||M^*||M_t^s$. It receives back $(IV, C_1, \ldots, C_t, T)$.

(3) The adversary $\mathcal{A}_{MAC}$ extracts the ciphertexts $(C_1, \ldots, C_q)$ and $(C_{q+r+1}, \ldots, C_t)$ and sends to $\mathcal{A}$ the values $(IV, (C_1, \ldots, C_q), (C_{q+r+1}, \ldots, C_t))$. Note, moreover, that the received values $(C_{q+1}, \ldots, C_{q+r})$ are in fact $\mathsf{AES}_K(ctr_{32}^1(V||0^{31}||1)), \ldots, \mathsf{AES}_K(ctr_{32}^{q+r}(IV||0^{31}||1))$, and it sends it to $\mathcal{A}$ as well as the key stream.

(4) When the adversary $\mathcal{A}$ queries its oracle F_{ObvPoly} on some polynomial $p^*(\cdot)$, the adversary $\mathcal{A}_{MAC}$ simply replies with the empty string λ.

(5) When the adversary $\mathcal{A}$ outputs some $(IV', C_1', \ldots, C_t', T'')$ to be delivered to the server, the adversary just outputs this message.

The adversary $\mathcal{A}_{MAC}$ perfectly simulates the view of the adversary $\mathcal{A}$ even without knowing the key K. Moreover, if the adversary $\mathcal{A}$ manages to output some ciphertexts $(IV', C_1', \ldots, C_t', T'')$ that is valid and is decrypted to messages that are not M_t^p, M_t^s – then this is a forge of a tag on some message that $\mathcal{A}_{MAC}$ did not query before. As a result, we conclude that the event in which the simulator has to verify that the simulated server outputs the same messages as the client sent is not necessary.

Hyb_2 and Hyb_3: In Hyb_2, the simulator uses the real inputs M_t^p and M_t^s and interacts with the adversary $\mathcal{A}$. The simulated server outputs some M_t^p, M^*, M_t^s in case of a valid execution, or $\perp$. In the former case, the simulator sends M^* as the input of the corrupted proxy, and the honest client in the ideal execution uses M_t^p, M_t^s. As a result, the output of the server is M_t^p, M^*, M_t^s, exactly as the simulated interaction. On the other hand, Hyb_3, the real execution, is exactly the simulated interaction performed by the simulator. We conclude that the outputs of Hyb_2 and Hyb_3 are identical. ■

This completes the proof of Theorem 8. ■