Secure Channel Injection and Anonymous Proofs of Account Ownership

Liang Wang
University of Wisconsin – Madison
liangw@cs.wisc.edu

Rafael Pass
Cornell Tech
rafael@cs.cornell.edu

abhi shelat
University of Virginia
abhi@virginia.edu

Thomas Ristenpart
Cornell Tech
ristenpart@cornell.edu

***Abstract*—**

We introduce secure channel injection (SCI) protocols, which allow one party 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. This allows alice@mail.com to prove ownership of some email address @mail.com, without revealing "alice" to the verifier. We show experimentally that our system works with standard email server implementations as well as Gmail. We go on to extend our basic SCI protocol to realize a "blind" certificate authority: the account holder can obtain a valid X.509 certificate binding alice@mail.com to her public key, if it can prove ownership of some email address @mail.com. The authority never learns which email account is used.

I. Introduction

In this paper we introduce and construct *secure channel injection* (SCI) protocols. These enable one party, *the client*, to securely allow another party, *the proxy*, to inject data into an encrypted channel between the client and a server. During the protocol execution, the client should learn nothing about the injected data and the proxy should learn nothing about other content of the encrypted stream.

To motivate SCI, consider the following scenario. The owner of an email account alice@mail.com wishes to obtain a certificate binding her email address to a public key. In existing systems, she must reveal her identify to a certificate authority (CA) service in order to prove ownership of the email address. This has privacy implications for Alice, as she may not want to trust the CA with her identity (c.f., [35]). For example, if certificates are to be used with end-to-end private messaging systems, there may be no reason a priori the CA should need to know Alice's exact identity. We call a CA blind if it can deliver certificates after checking ownership of an identity, while never learning the exact identity.

Unfortunately, no existing mechanisms allow building a blind CA. Traditional CAs use proofs of ownership of email (or other) accounts in the following way: the user, as prover, submits an email address and the verifier sends an email containing a random challenge to the address. Obtaining access to the challenge proves ownership of the account. Anonymous credential systems (e.g., [9], [38]), allow proving one has a credential without revealing it, but these systems nevertheless require a traditional registration step and proof of ownership.

We show how to build *anonymous* proof of account ownership (PAO) that allow proving ownership of an existing email account without revealing which account. To do so, we introduce the more general tool of secure channel injection (SCI). As mentioned, an SCI protocol allows a client and proxy to jointly generate a sequence of encrypted messages sent to some server, with the ability of the proxy to insert a private message at a designated point in the sequence. In our proof of ownership context, the server would be an authenticated email service, and the injected message will be a challenge inserted into an email. To complete the proof, the client can later retrieve the challenge from the service using a separate connection.

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 special-purpose secure two-party computation protocols that allow the client and proxy to efficiently compute a TLS session with the server. For most of the session, the proxy acts as a simple TCP-layer proxy that forwards messages back and forth. The client negotiates a TLS session key directly with the destination server. At some point in the stream, however, the proxy must inject a message, and here the client (which has the session key) and the proxy (which has the secret message 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 end up with a protocol whose most expensive step is a standard Yao [49] two-party protocol on a circuit consisting of one to two AES computations (plus exclusive-or operations). Direct use of Yao to perform the entire record layer construction without our tricks would be prohibitively expensive.

Our SCI construction can be used to perform anonymous PAO. The proxy can be directed to connect to an authenticated SMTP service, and then run the SCI to send an email from the client's email account to some other (possibly the same) account from which the client can read received email. The SCI injects into the body of the email a challenge, and the client proves the email gets sent by obtaining the challenge later and sending it back to the proxy.

We go on to show how to extend our basic SCI-based anonymous PAO protocols for TLS to additionally yield as output a cryptographic commitment by the client to the account identity, i.e., the sending email address. This commitment can be used in conjunction with the challenge in a subsequent step to obtain a certificate from the proxy that binds the account identity to a public key of the client's choosing. Thus we obtain the first construction of a blind CA: it checks a user's

ownership of an email address and, if successful, generates a signed X.509 certificate for them, without ever learning the identity of the user. The identity provider (the email service) need not be modified, and never even learns that they are being used as an identity provider.

We implement a prototype of the functionalities above and test it with various SMTP servers, showing that it is fast enough for deployment.[1] Running the client on a laptop connected via a public wireless network to a proxy running on EC2, the median time to complete a SCI-based anonymous PAO is 1.6 seconds. 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 secure channel injection to build anonymous proof of account ownership protocols for SMTP-STARTTLS.
- We show how to extend our protocols to enable the construction of a blind CA that will generate a certificate binding a public key to an account identifier only should ownership of the account be proven, all without having the CA learn which account was used.

II. Overview

In this paper we introduce and construct *secure channel injection* (SCI) protocols. These enable one party, the client, to securely allow another party, the proxy, to inject data into an encrypted channel between the client and a server. The client should learn nothing about the injected data and the proxy should learn nothing about other content of the encrypted stream. We believe SCI protocols to be of potentially broad applicability in privacy-sensitive settings. We now illustrate one example application: anonymous proof of account ownership (PAO).

Proofs of account ownership. A conventional proof of email ownership works as follows. The alleged owner of an email address, say `alice@gmail.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@gmail.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.) 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).

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 web sites can be proven by adding a webpage that contains a challenge value, and similar approaches work with Twitter and Facebook accounts [29]. 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 [27], which provides free TLS certificates to users that can prove ownership of the domain via a DNS proof of ownership or web page proof of ownership. Keybase.io signs PGP keys based on proofs of ownership of social media accounts [29]. 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 account owner / prover may be unwilling to reveal their identity. In the end-to-end encryption setting, privacy is an often mentioned critique of certificate transparency mechanisms like CONIKS [35]. 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 [9], [25], [38], [39]. However, these systems are vulnerable if the TTP misbehaves.

We introduce the idea of anonymous proofs of account ownership. Instead of proving that the account owner owns a particular email address such as `alice@gmail.com`, the verifier will be convinced that the prover owns some valid email addresses at Gmail. To accomplish this we will need the following from the service (Gmail in our example):

- The service exposes an authenticated API.
- That authenticated API allows submission of content that can be read only by the account owner.
- The API is over an encrypted channel.

The second criteria turns out to be tricky, as API implementations often have subtleties or side-effects that might undermine a anonymous PAO scheme. For example, while in theory one could use as target service sites like Facebook and Twitter, their APIs are often insufficiently understood to ensure security.

Anonymous PAO needs to provide *validity* and *anonymity*. The first asks that at the conclusion of the protocol, it should not be possible for a prover to convince the verifier if in fact the prover does not have the ability to log into the service with a valid account. Anonymity requires that the verifier not be able to identify which account was involved in the API requests.

[1] We plan to make our implementation public and open source.

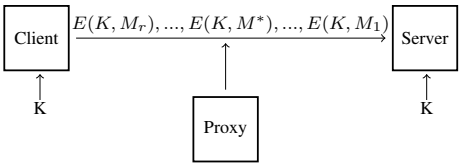


Fig. 1: A high level view of SCI.

In some contexts we might want an additional property that we call *service obliviousness*. This is achieved if the service cannot detect that a particular API access was used to perform a proof of ownership.

We will also show how to use anonymous PAOs 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.

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. 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 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 only explore use of SCI protocols in the context of anonymous PAOs, other applications might surface in future work.

III. Secure Channel Injection

For the purposes of description it suffices to give a simplified view of secure channel (SC) protocols. Let $\mathcal{SC}$ 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.

A secure channel injection (SCI) protocol is parameterized by a secure channel protocol $\mathcal{SC}$, a target message index $t \in [1..r]$, and an injection template M_t. The target index t indicates which client-to-server message will have content injected by the proxy. The template $M_t = M_t^p \| \tilde{M} \| M_t^s$, where $\tilde{M}$ is a placeholder for an injected message, and M_t^p and M_t^s are the template prefix and suffix respectively. M_t specifies the injected message as well as what data format the injected content should be. We refer to M^* as the injected message. We assume that $\mathcal{SC}$ is such that computing the messages $M_{t+1}, \ldots, M_r$ following the injected message can be done without knowing M^*. A SCI protocol is then defined by by client and proxy algorithms that we denote by $\mathcal{P}_{cl}$ and $\mathcal{P}_{pr}$, respectively.

An SCI protocol $\mathcal{SCI}$ is correct if execution of the client and proxy with a server implementing $\mathcal{SC}$ results in the transcript $M_1, \ldots, M_t^p \| M^* \| M_t^s, \ldots, M_r$ observed by the $\mathcal{SC}$ server.

To make this concrete, our implementation later targets $\mathcal{SC}$ being TLS (version number 1.1 or greater), and E being the TLS record layer MAC-Encode-Encrypt construction with HMAC-SHA256 and AES-CBC.

Threat model. In terms of security, an SCI protocol should achieve four basic security goals:

(1) *Injection secrecy*: The client cannot learn M^* during the protocol interactions.[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*: The server cannot distinguish an SCI execution from a standard execution of a client.

Achieving these security properties will only be possible in the case that the service, client, and proxy do not pairwise collude. If the service and client collude, then injection secrecy is impossible. If the service and proxy collude, then transcript privacy, transcript integrity, and server obliviousness are impossible. If the client and proxy collude, the whole setting becomes meaningless.

We will target protocols that achieve the first three security goals even when one of the parties, either the client or the proxy is malicious, meaning they can deviate arbitrarily from the protocol execution. We also consider a weaker goal in which the client or the proxy is an honest-but-curious adversary, i.e. all parties faithfully execute the $\mathcal{P}_{cl}$ and $\mathcal{P}_{pr}$ protocols, but then inspect the transcripts to learn extra information about the private inputs or outputs. For example, the client might try to violate injection secrecy by studying the transcript of its interactions with the proxy. Likewise the proxy might try to violate transcript privacy using the transcript of its interactions with the client and server.

Finally, we model the server as an honest-but-curious adversary, meaning it might try to violate server obliviousness by inspecting the sequence of packets sent to and from it and the sequence of plaintext messages $M_1, \ldots, M_r$, but it won't maliciously deviate from its normal functionality. We assume that the proxy IP does not, by itself, suffice to violate server obliviousness.

We 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, while the client cannot access the network transcripts between the proxy and the server.

Other types of attacks that are not directly related to 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.

IV. SCI for TLS

As mentioned in §III, we focus on implementing SCI using TLS as the underlying communication protocol. In this case, $E(K, M)$ represents a TLS record layer encryption. While there are several options supported in the wild, we focus on the currently commonly used one in TLS 1.1 and 1.2: AES using CBC with HMAC SHA-256 in the MAC-Encode-Encrypt mode of operation. As in Figure 2, we need to design a protocol that allows two parties to jointly compute such a

[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.

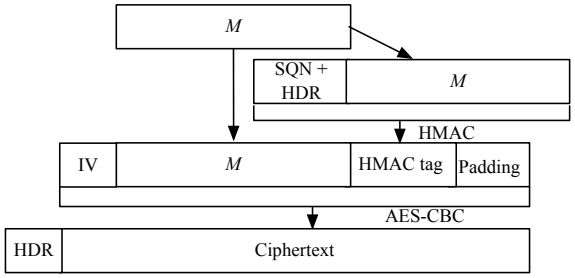


Fig. 2: Record construction procedures for a message M in TLS (version $\geqslant$ 1.1). HDR is a 5-byte TLS record header, and SQN is a 8-byte sequence number. IV has a fixed size of 16-byte. The size of the HMAC tag depends on the hash functions being used in HMAC. Before AES encryption, the record will be padded to a multiple of 16 bytes [15], [16].

TLS record, with one party providing the TLS session key and part of the message, and the other one providing the rest of the message.

One solution is to directly use secure multi-party computation (SMC) to let two parties compute the TLS record. However, using common SMC techniques, such as Yao's protocol [49] or fully homomorphic encryption [21], would be prohibitively 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. Our more efficient protocol, named SCI-TLS, boils down to having to use general-purpose SMC only on invocations of AES. In this section, we first introduce two protocols called 2P-HMAC and 2P-CBC, and design SCI-TLS based on the two protocols. Then, we analyze the security of the designed protocols.

A. 2P-HMAC and 2P-CBC

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 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. We require that M_t^p is a multiple of d bits in length during HMAC computation (or n bits during CBC), so is M^*.

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 and outputs an 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 $\text{PadH}_{|M|}$ so that $M \| \text{PadH}_{|M|}$ is a multiple of d bits. For SHA-256 for example $\text{PadH}_\ell = 10^r \| \langle \ell \rangle_{64}$ 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 $H(M) = f^+(\text{IV}, M \| \text{PadH}_{|M|})$.

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

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

where $ipad$ and $opad$ are the inner and outer padding constants each of length d bits [31]. 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 Figure 3. We denote an execution of this protocol by $\text{2P-HMAC}((K_{hmac}, M_t^p, M_t^s), M^*)$. We will discuss the security of this protocol later in this section.

Cl: K_{hmac}, M_t^p, M_t^s		Pr: M^*
$s_0 = f(\text{IV}, K_{hmac} \oplus ipad)$		
$s_1 = f^+(s_0, M_t^p)$	$\xrightarrow{s_1}$	$s_2 = f^+(s_1, M^*)$
	$\xleftarrow{s_2}$	
$y = f^+(s_2, M_t^s \| \text{PadH}_\ell)$		
Output $H\big((K_{hmac} \oplus opad) \| y\big)$		

Fig. 3: The 2P-HMAC protocol. The value ℓ is the length of $M_t^p \| M^* \| M_t^s$ and we assume this is known to both parties ahead of time.

2P-CBC. We now turn to how to jointly compute a CBC encryption over $M_t^p \| M^* \| M_t^s$. For an n-bit string S and a string M consisting of m n-bit blocks $M_1, \ldots, M_m$, we define $\mathsf{CBC}(K_{aes}, S, M)$ to output $C_1, \ldots, C_m$ where $C_0 = S$ and $C_i = \text{AES}(K_{aes}, M_i \oplus C_{i-1})$ for $i = 1$ to m. Then CBC mode on message M is defined by choosing a random n-bit IV and computing $\mathsf{CBC}(K_{aes}, \text{IV}, M \| \text{PadC}_{|M|})$ where $\text{PadC}_{|M|}$ is a padding string defined solely by n and the length $|M|$. It ensures that $M \| \text{PadC}_{|M|}$ is a multiple of n bits in length.

To jointly compute CBC mode, the client picks an IV and computes CBC over the template prefix M_t^p and suffix M_t^s, which we assume must have length a multiple of n bits. Let the prefix of the ciphertext be $C_0, \ldots, C_p$. Then the two parties can use a general-purpose SMC protocol to compute $\mathsf{CBC}(K_{aes}, C_p, M^*)$ where K_{aes} and C_p are the private inputs of Cl and M^* is the private input of Pr. Again we assume $|M^*| = j \cdot n$ for some small integer j. The output of the SMC is $C_{p+1}, \ldots, C_{p+j}, C_{p+j}$. If $j \geqslant 2$, the first $j - 1$ blocks are given to the proxy and the last is given to the client. Having C_{p+j} allows the client to compute the suffix of the encryption locally via $\mathsf{CBC}(K_{aes}, C_{p+j}, M_t^s)$. If $j = 1$, 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 simply require $|M_t^s| = 0$; that is, $|M^*|$ is the last

[3] Our protocol here will not work with SHA-3 whose compression function is not secure (e.g, Keccak, who uses sponge construction [6]). This is related to so-called mid-game attacks [10].

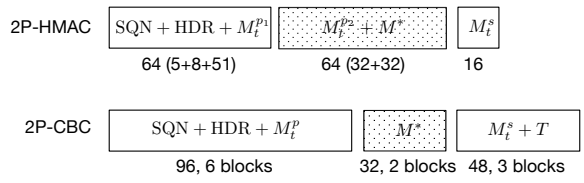


Fig. 4: An example of injecting a 32-byte M^*. The messages with dots are input by the proxy, and the other message are provided by the client. The numbers under each messages are the message sizes (in bytes) and the number of AES blocks.

block of the plaintext. Then, the proxy doesn't need to send back C_{p+1}.

We let $\mathsf{2P\text{-}CBC}((K_{aes}, M_t^p, M_t^s), M^*)$ denote the just described protocol.

B. The SCI-TLS 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 $P_1, \ldots, P_t^*, \ldots, P_r$ be that sequence of TLS plaintext fragments sent from the client to the server in separate record layer encryptions, with P_t^* representing the fragment within which the proxy will inject its private message M^*. The corresponding plaintext of P_t^* is $M_t^p \| M^* \| M_t^s$, that both $|M_t^p| + 13 \cdot 8$ and $|M^*|$ are multiples of both d (512 bits) and n (128 bits). The value 13 comes from the 13-byte header (*SQN* and *HDR*) during HMAC.

Some applications may want to inject less than d bits of data, and thus are not able to satisfy the requirements that M_t^p ends on an HMAC block boundary and that $|M^*|$ is a multiple of d. To alleviate alignment problems and make the injection more flexible, we let $M_t^p = M_t^{p_1} \| M_t^{p_2}$ where $M_t^{p_2}$ can be shared with the proxy. Then, the $|M^*|$ can be a multiple of n. For example, to inject a 128-bit M^*, $|M_t^{p_2}|$ can be 384 bits. In fact, in several of our applications both M_t^p and M_t^s can be made public to the proxy and only *other* TLS plaintext fragments must be kept secret.

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 $t-1$ TLS record layer fragments. Let the client-to-server session keys be K_{hmac} for HMAC and K_{aes} for AES. To send P_t^* the client constructs the message prefix $SQN \| HDR \| M_t^p$. (Recall from Figure 2 that TLS records include a header and sequence number.) Then Cl and Pr execute

$$\mathsf{2P\text{-}HMAC}((K_{hmac}, SQN \| HDR \| M_t^{p_1}, M_t^s), M_t^{p_2} \| M^*)$$

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. An example is shown in Figure 4. If $|M^*|$ is 128 bits (corresponding to $j = 1$ in §IV-A), they execute

$$\mathsf{2P\text{-}CBC}((K_{aes}, M_t^p, M_t^s), M^* \| T)$$

where $|M_t^s| = 0$ and T is shared with the proxy by the client. We omit paddings in the above descriptions. That ciphertext (with *HDR* being added) then gets sent by the proxy. Finally the proxy can resume being a TCP-layer proxy for subsequent record layer encryptions.

C. Security Analysis

Recall that we seek injection secrecy, transcript privacy, transcript integrity, and server obliviousness. We discuss each in turn, assuming 2P-CBC uses a maliciously secure SMC. We discuss honest-but-curious SMC at the end of the section.

Injection secrecy. Can a malicious client learn anything about the injected message M^*? During 2P-HMAC the client sends the proxy a chaining variable s_1 of its choosing and receives back the value $s_2 = f^+(s_1, M^*)$. When f is a fixed-input length RO, results of [11] imply that f^+ on a fixed number of blocks is indifferentiable from a RO. This implies that given s_1 and s_2, the malicious client can determine M^* only via queries to a random oracle and so one gets security proportional up to $q/2^\mu$ for q queries and where $\mu = -\log \max_X \Pr[M^* = X]$ is the min-entropy of the distribution from which M^* is drawn. Looking ahead, we will ensure in applications that $\mu \geq 128$.

With respect to 2P-CBC, the SMC protocol, if it is secure against malicious clients, leaks no information about M^* other than what is implied by the output. But the client's output does not include the ciphertext blocks associated with M^*, meaning the output of the SMC leaks nothing about M^* (see §IV-A and §IV-B). A malicious client can submit $C_p \oplus \Delta$ for some fixed offset Δ, thereby changing the value M^* received by the server. This opens up the possibility that the client could attempt to learn something about M^* by observing how the remote server reacts to received messages. In applications M^* will be a data value, but we will anyway take care to ensure such malicious modifications do not give rise to side-channels.

Transcript privacy. All plaintext fragments beyond P_r^* in SCI-TLS are protected by the TLS record layer and so the proxy cannot learn anything about them assuming secure TLS implementations are being used on the client and server. That leaves the client's portions M_t^p and M_t^s of P_r^*. The 2P-HMAC protocol sends the proxy $s_1 = f^+(f(\text{IV}, K_{hmac} \oplus ipad), M_t^p)$ but K_{hmac} is a uniform secret unknown to the proxy. Assuming f is a random oracle, s_1 leaks nothing about M_t^p with probability about $q/2^{|K_{hmac}|}$ for q queries. The 2P-CBC, if using a maliciously secure SMC, does not leak anything about the n bits of M_t^s — the output C_{p+j+1} eventually seen by the proxy is ciphertext.

Transcript integrity. At first glance, 2P-HMAC is susceptible to a kind of length extension attack [2], since we allow the proxy to append an arbitrary message. However, the client has indirect control over the length of M^*, because it computes the final value over PadH which contains the length of the total message. For 2P-CBC, the length of blocks is fixed and the client can abort if the proxy attempts to deviate from this. If the proxy inserts different data into 2P-HMAC than 2P-CBC (e.g., the proxy inputs an arbitrary T in 2P-HMAC when $|M^*|$ is 16 bytes), the server will reject because the HMAC will not verify properly.

Server obliviousness. We do not always need server obliviousness, but some applications may require it and so we strive to achieve it. The SCI-TLS protocol effectively implements a TLS client (jointly computed by the client and proxy). A server

could, however, attempt to fingerprint this implementation, for example if we used esoteric ciphersuites that lead to fingerprintable handshake messages. This is a challenge for our implementation, and we took care to mimic a popular TLS implementations (Firefox [46]).

Traffic analysis (e.g., timing of messages) could be used by the server to try to detect SCI-TLS, and this particularly seems dangerous given the more complex interactive computation of records. We provide a preliminary investigation into this issue using our prototype implementation, see §VII.

The IP address of the proxy may cast suspicion on the connection, either because it is known to be associated with a SCI-TLS service or because the client has not logged in from this location previously. Deployment would need to obtain new IPs for proxies and prevent enumeration (e.g., by spawning new proxies on demand). To deal with the second issue, one would have to ensure at least that the IPs are plausible (e.g., not from a strange country).

Honest-but-curious SMC. Rather than a maliciously secure SMC, one might want to use an honest-but-curious SMC protocol within 2P-CBC. This means it is only secure against attackers that follow the protocol. Accordingly, a malicious client could violate injection secrecy or a malicious server could violate transcript privacy (though just for M_t^p and M_t^s). The other security properties seem to hold regardless in the presence of malicious attackers despite using an honest-but-curious SMC.

D. Other Secure Channels

We focused above on TLS using a common record layer encryption scheme. Our techniques can be adapted to some other cipher suites and protocols, but not others.

In the first category, a popular approach to building authenticated encryption is to first encrypt the plaintext and then authenticate it using a MAC. If one uses CTR mode encryption (or any other stream cipher or stream-cipher like mode) followed by HMAC, one can build a very efficient SCI protocol. The client can just send the proxy the key stream needed to encrypt the injected message. Then, the parties can use 2P-HMAC to jointly compute the tag value. This would be very efficient. Unfortunately TLS does not have a cipher suite standardized using CTR mode plus HMAC. IPsec does support it.

If one uses CBC mode instead of CTR mode (as done in SSH [50]), then one could use 2P-CBC followed by 2P-HMAC to perform the injection.

There are a number of in-use authenticated encryption schemes such as GCM [34], ChaCha20/Poly1305 [37], and CCM [33] for which we do not yet know 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 or universal-hashing. These unfortunately do not seem secure should one try to split computation of the MAC as in 2P-HMAC, in particular the client might be able to violate injection secrecy. The reason is that their compression function analogue does not have any cryptographic strength to a party that knows the secret key. We leave finding efficient SCI protocols for such schemes as an open problem.

V. Anonymous PAOs and Blind Certificate Authorities

We introduce anonymous proofs of account ownership (PAOs) in this section. Unlike conventional PAOs in which a verifier sends a challenge in a message to a prover's account, our anonymous PAO realizations will instead require the prover to send an SCI-injected challenge message using her account. The challenge message can be sent to any location only accessible to the prover who can then retrieve the challenge and send it to the verifier. Assuming the server authenticates message requests, this will prove ownership of an account.

We then show how to extend our SMTP anonymous PAO to build a blind certificate authority (CA) 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 will bind the public key to the email account. Unlike conventional CAs, ours will be *blind*: the CA doesn't learn the identity or public key of the user. The certificate can be later used for, e.g., secure end-to-end encryption for email or messaging. Or it can be used within an anonymous credential system [9], [25], [38]. We emphasize that in existing anonymous credential systems, users always must reveal themselves during registration.

We focus on one application layer protocol, SMTP, and briefly discuss challenges to working with other target protocols towards the end of the section.

A. SMTP with STARTTLS

Our protocol is intimately tied to the workings of SMTP implementations, so we first discuss the workflow of sending an email in SMTP-STARTTLS. We here focus on PLAIN as the target authentication mechanism, which is the most widely used authentication mechanism as reported by [26]. To start a TLS-protected SMTP session, a client first issues a `EHLO` command in cleartext to check if the target SMTP server supports `STARTTLS`, and sends a `STARTTLS` command to start a TLS connection if the server does. Then, the client will send an `AUTH PLAIN user password` command to the server, and the username and password are BASE64-encoded. If the authentication is successful, the client further sends `MAIL` and `RCPT` commands to set the email addresses of the sender and the recipient respectively, followed by a `DATA` command to notify the server the start of email transactions. Finally this is followed by the email content. After building the TLS connection, the four commands (`AUTH`, `MAIL`, `RCPT`, and `DATA`) are mandatory, and must be sent in order to successfully send an email, according to RFC 5321 [30]. Other commands in SMTP are optional during the SMTP session. Finally, the client can send the `QUIT` command to finish the session.

The SMTP server might support the `PIPELINING` extension. If the `PIPELINING` extension is enabled, a user can send a group of commands in a single message without waiting for the responses from the server.

B. Requirements for SMTP Servers

We assume the target SMTP server is correctly configured as a closed relay to avoid being abused for spamming (e.g., it rejects emails from local users with incorrect or spoofed sender addresses). Thus only authenticated users can send email using the SMTP server. Also we assume the server follows RFC standards and best practices [18], [32], [48].

In order to run blind CA successfully, it's important for the CA to determine if: (1) the server echos back received commands to the client — this would immediately break injection secrecy; (2) if the server supports the `PIPELINING` extension; and (3) if the server is RFC-compliant in terms of how it respond to pipelined commands.

One can check if (1) holds by sending an invalid command (such as "AAA") to the target server and seeing if the response contains the invalid command. For (2), one can simply send a `EHLO` command to check the server's supported extensions. For (3), the CA can send a sequence of `EHLO` commands in one message and count the number of responses. According to RFC, the server should send responses for each command separately. In some cases, one can also obtain further information about the service by issuing a `HELP` command to determine the SMTP software version, but this is not always available due to security concerns by administrators.

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 [1], [19]. They appear in a wide variety of ISPs and email services. Using a lightweight SMTP client, we probed each server to determine if they satisfy the three requirements above.

We find only 11 of 150 SMTP servers echo back commands, making the rest potentially suitable for use with our protocol. We also discovered that all 150 servers use *sendmail* [42]. We find 61 (41%) of 150 SMTP servers do not support pipelining, which suggests that disabling pipelining is a common configuration in the real world. Though pipelining can reduce SMTP transaction time, spammers can leverage it to send bulk messages without respecting the response codes, and thus overwhelm the SMTP server [47]. That is one possible reason for disabling pipelining. In those SMTP servers that support pipelining, 51 of them are RFC-compliant.

Overall, we find 112 SMTP servers (75% of those examined) meet our requirements, including popular services that have a large user base: Gmail, Outlook, Hotmail, Mail.com, etc. In deployments of our protocol one would need to perform similar scans before use to ensure our assumptions hold. We will release a tool with our prototype to help to determine if a SMTP server can be used with anonymous PAOs or blind CAs.

C. Anonymous PAO for SMTP

We now describe how to use our SCI-TLS protocol to perform an anonymous PAO for SMTP-based services. One can prove to the verifier that she owns an email account from a certain service, without disclosing the exact email address. The three parties in this case are: (1) a prover who owns an email account `user@domain.com`; (2) the service `domain.com` that administers `user@domain.com`; and (3) a verifier that will check the prover's ownership of the email address. We assume for simplicity that the verifier is synonymous with the proxy we will use for SCI. The SCI client is the prover, and the server is the service. We also assume that the client might communicate with the proxy through an anonymous or pseudonymous channel, such as Tor and public wireless networks, to achieve IP anonymity.

SCI-based anonymous PAO. The setup is shown in Figure 5: the prover runs a modified SMTP client, and uses SMTP over TLS to interact with the private SMTP server of `domain.com` via a proxy managed by the verifier. The proxy is responsible for determining the address of `smtp.domain.com`, e.g., via DNS, and by our assumptions above this is sufficient to uniquely identify the server as only accepting authenticated clients for `domain.com`. The goal will be to send an email from `user@domain.com` to an email account accessible to the prover[4], with a body that contains a secret challenge injected by the proxy.

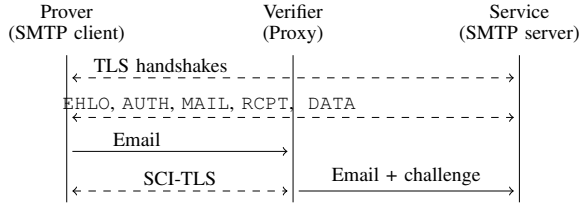
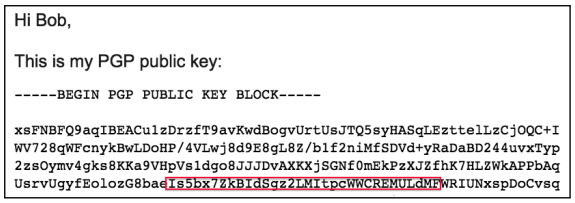


Fig. 5: Basic setup for SMTP-based PAO


Hi Bob,

This is my PGP public key:

-----BEGIN PGP PUBLIC KEY BLOCK-----

xsFNBFQ9aqIBEACulzDrzfT9avKwdBogvUrtUsJTQ5syHASqLEzttelLzCjOQC+I
WV728qWFcnykBwLDoHP/4VLwj8d9E8gL8Z/b1f2niMfSDVd+yRaDaBD244uvxTyp
2zsOymv4gks8KKa9VHpVs1dgo8JJJDvAXKXjSGNf0mEkPzXJZfhK7HLZWkAPPbAq
UsrvUgyfEolozG8baeIs5bx7ZkBIdSgz2LMItpcWWCREMULdMFWRIUNxspDoCvsq

Fig. 6: A screenshot of the message template for SMTP PAOs and the challenge (red rectangle). We only show a partial message template to save space.

The prover uses the modified SMTP client to start a SMTP (over TLS) session to the SMTP server of `domain.com`. Most of the SMTP messages are sent in separate TLS fragments because of the interactive nature of SMTP, including the sender and recipient. So the client signs into the server with the user's account, and sets the necessary information (recipient address, subject, etc.). Immediately following the `DATA` flow, the body of the message must be sent. For this 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).

The TLS fragment containing the content is handled via our SCI-TLS injection protocol; this is the P_t^* message using the notation from §IV. The resulting TLS record will be sent to the server by the verifier's proxy, and after this the client can send the `QUIT` command to end the session. Later, the prover checks the recipient's email to find the challenge, and reveals it to the verifier to complete the proof.

To use the target SMTP server to send emails, the prover must have a registered email account and be authenticated. The fact that she is able to send the email and obtain the challenge later proves ownership of some email account from `domain.com`. SCI-TLS injection secrecy plus requirements on the SMTP server ensures the prover can't cheat by learning the challenge via the protocol execution. Transcript privacy

[4]One could have the email sent to `user@domain.com`, in fact.

ensures that the sender and recipient of the email are hidden from the proxy.

Challenge steganography. Recall that one of our security goals is service obliviousness. This isn't always important, but could be in some settings (see our example below). Assuming that SCI-TLS is server oblivious, meaning it isn't 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, in SMTP-based PAO, 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. In our PAO prototypes, we build a rudimentary template generator that can be easily used to generate new templates, based on given parameters. One example of generated message template is shown in Figure 6. That said, good steganography is tricky and future work could help build better message template generation tools.

Example use case: whistleblowing. This work was originally motivated by the problem of verifying a whistleblower's access to an organization while communicating electronically with a journalist. This is a situation in which service obliviousness is paramount: after a leak there may be an investigation in order to identify the whistleblower. While listening to the story from a whistleblower, a journalist may want to have additional information that can be used to verify the identity of the whistleblower. As for the whistleblower, she may also want to provide the necessary information that helps to convince the journalist of her story; but no information should be linked to, or can be used to discover, her identity. Usually, an employee from a certain organization will have either an working email address. In this case, the whistleblower can use SMTP-based PAOs to provide some evidence of insider access to the organization.

D. Blind Certificate Authorities

The basic idea is to use our SMTP anonymous PAO protocol, but additionally output from it a cryptographic commitment to the user's email account `user`. If we can obtain a secure commitment from the SCI-TLS protocol to the user's email account, then it can be used in a second phase to run a specially constructed protocol with the CA that: (1) verifies that the commitment can be opened by the client to the same email account as that in the certificate and (2) produces a hash of the certificate without revealing to the CA its contents. The CA can sign the hash using a standard digital signature scheme in order to finalize the certificate.

Anonymous SMTP-PAO with commitment. As described in §V, in a SMTP-STARTTLS session four commands (`AUTH`, `MAIL`, `RCPT`, and `DATA`) are mandatory. Our SMTP client for the user is written as only using the minimum number of commands to send an email.

Excepting `AUTH PLAIN`, each command and the email contents will be sent in one TLS fragment. The `AUTH PLAIN` command needs to be split into two fragments,

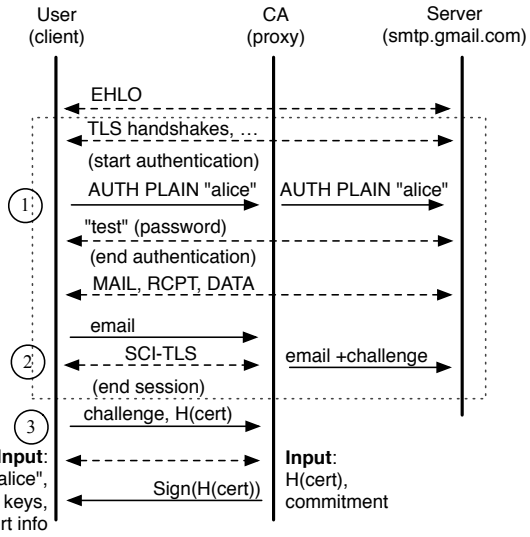


Fig. 7: Blind CA workflows. The communications in the dash rectangle happen in the same TLS session. ① The CA saves the first message as the commitment; ② The CA uses anonymous PAO to send a challenge; ③ The CA and the user run a zero-knowledge protocol to generate a legitimate X.509 certificate.

one for `AUTH PLAIN [sep]user[sep]` and one for `password` (`[sep]` is a separator defined in [30] to separate command and parameters). Thus the first message from the client will contain the client's email account, and the fifth message will be the email contents into which a challenge will be injected. We can split commands into multiple messages because a command in SMTP is terminated by the line terminator `CRLF` and SMTP servers will buffer messages until it sees the line terminator, combine the buffered message as a command, and clear the buffer.

In summary, the client and proxy (run by the CA) run SCI-TLS for SMTP where the injection will be in the fifth message. The first ciphertext sent from client to server, which is the TLS encryption of the client's email account (plus a command string, spaces, and two separators), will be taken as the commitment C. The opening of the commitment is the associated HMAC key K_{hmac} and AES key K_{aes}, and the client stores these for later use. The challenge to be injected is set to be a random value M^*, and the proxy adds C to a table under index M^*.

While running SCI-TLS, the proxy expects to see exactly six messages (four commands plus an email) sent from the client, and five responses from the server (not counting TLS handshake messages and the `EHLO` at the very beginning). It is important that the proxy abort after seeing 6 messages. If it allows, more attacks arise, as we discuss later.

Some servers might require the `EHLO` after `STARTTLS`. One can check this by sending three commands (`EHLO`, `STARTTLS`, and `AUTH PLAIN` in order and see if the server asks for an extra `EHLO`. In this case, the proxy simply needs to let the client use seven messages (with six responses) to

finish the session, and grabs the second message seen as the commitment.

Certificate generation. Assuming the SCI-TLS injection is successful, the client can retrieve M^*. It sends this to the proxy as well as a hash $h = H(cert)$ of its certificate $cert$, with the *subject* field being set to the client's email account `user`. The proxy retrieves the commitment C based on M^*. Now the client and proxy engage in a zero-knowledge protocol in which the client demonstrates knowledge of (a) information necessary to form $cert$, including the client's email account, and (b) the keys K_{hmac}, K_{aes} used during the PAO session in which the email account was used to send an email.

In detail, the public statement includes h and the commitment C. The client has a private witness consisting of $cert$, email account, and the opening K_{hmac}, K_{aes}. The zero-knowledge proof verifies that client knows an opening of C to retrieve a message within it, which, when incorporated into the certificate $cert$, produces the hash h. In this way, the proxy can be certain that h is computed with respect to the same email account as was used in the anonymous PAO protocol.

In practice some fields of certificates are set or checked by the CA (e.g., expiration duration policies). Set values can be sent to the client before generating the cert. The certificate can be checked during the zero-knowledge proof with suitable extensions in the computed function.

Assuming the outcome of the zero-knowledge proof protocol is accepted by the CA, then the CA can sign the hash value h and send the result back to the client.

The above approach ensures that the zero-knowledge proof only operate over symmetric cryptographic operations (hashing and block ciphers). We can use recent advanced techniques [20], [22], [28] for constructing fast zero-knowledge proofs over boolean circuits to make this protocol efficient. This enables efficiency in practice. Another approach would simply be to perform SMC to directly produce a blind signature of the cert, but this would require performing asymmetric operations within SMC, which is prohibitive.

E. Security Analysis

Protecting against injection attacks. The proxy might attempt to violate transcript privacy and user 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. The proxy then can obtain the email address of the client as well as partial email contents. But this attack doesn't work if the target server disable pipelining, hence our requirement to this effect discussed in §V-B.

There are other ways one could potentially circumvent this issue, if one wants to work with SMTP servers supporting pipelining:

(1) In 2P-CBC, the client checks if the input M^* from the proxy contains certain keywords using SMC before encryption. This requires a change to the SMC circuits that will increase complexity, and thus reduce anonymous PAO performance.

(2) In 2P-CBC, the client applies a simple encryption function to M^* before AES encryption. For example, the client can XOR M^* with a random string or shuffle M^*. In SMTP PAOs, to make sure the function output is still in valid Base64 format, the client can choose the function as byte-level shuffling. This is easy to achieve and wouldn't cause too much performance loss in 2P-CBC. However, we have to change 2P-HMAC to use SMC in order to let the client apply a function to M^*, which will reduce the PAO performance significantly.

(3) We restrict the length of the injection message to be 16 bytes. In this case, The mandatory fixed bytes needed (such as CRLFs, command keywords, spaces and newlines) in the commands are already more than 16 bytes, so the attacker will not be able to initiate a new email under this length restriction. This may reduce flexibility on the format of injected challenges, but 16 bytes suffices for the PAO applications.

Detecting protocol violation. If the SMTP server supports the `PIPELINING` extension, the user can send multiple commands in a single message. However, if the server is RFC-complaint, it will send responses for each command separately, and this will reveal the number of commands the client actually sends. If it detects that the user deviates from the agreed protocol (e.g., sending one message but receiving multiple responses), the proxy will terminate the session.

A malicious user might change the client program to attempt an identity binding attack, i.e., sending `AUTH PLAIN` α in the first message to get a certificate for α, where α is not owned by her. However, the user needs at least one extra round-trip if any command fails, making her unable to send an email using the required number of requests. Consequently, the user wouldn't be able to receive the challenge. The user might also try to split the `AUTH` command incorrectly so the username in the first message is just a prefix of her username (e.g., "ali" instead of "alice"). However, the message will be illegally formatted because it will miss the separator that is supposed to be at the end of the message (see §V-D). Such messages will be detected and rejected by the proxy when it verifies the zero-knowledge proofs.

TLS record as commitment. Overall, our restrictions guarantee that underlying the first TLS record layer encryption is the user's account and that this account was indeed authenticated by the server. It remains to show that the TLS ciphertext C is in fact a secure commitment. Authenticated encryption schemes are not, in general, good commitment schemes when using their secret keys as the opening. While they achieve hiding because of their confidentiality guarantees, most AE schemes suffer from trivial binding attacks. It happens that the MAC-Encode-Encrypt mechanism underlying TLS can be proven to be a hiding and binding commitment when modeling HMAC as a random oracle (an assumption justified in [17]). To sketch the proof we will argue that with overwhelming probability no attacker can find two key pairs $(K_{hmac}, K_{aes}) \neq (K'_{hmac}, K'_{aes})$, two plaintexts $M \neq M'$, and an IV such that the encryption of M under the first key pair and the IV yields the same ciphertext as the encryption of M' under the second key pair and the IV. This would rule out binding attacks because no single ciphertext can be decrypted under distinct keys to distinct messages.

Consider any fixed ciphertext associated to M. Then there are at most 2^{128} keys under which the ciphertext can be decrypted, with each giving rise to an associated message and tag $M'_i \| T'_i$ for $1 \leq i \leq 2^{128}$. Recall that the length of these tags is 256 bits. The goal of the adversary is to find an HMAC

key K'_{hmac} for which $\mathsf{HMAC}(K'_{hmac}, M'_i) = T'_i$ for some i. Fix some i; each random oracle query made by the adversary has a $1/2^{256}$ chance of hitting any particular T'_i. Thus, as long as the adversary makes less than 2^{80} queries, by a union bound over the queries, the probability that any such query hits T'_i is less than $2^{80} \cdot 2^{-256} = 2^{-176}$. It follows by a union bound over all "possible" i that the probability of there existing some i such that the adversary queries the random oracle on an HMAC key K'_{hmac} such that $\mathsf{HMAC}(K'_{hmac}, M'_i) = T'_i$ is less than $2^{128} \cdot 2^{-176} = 2^{-48}$. This proof approach leads to a loose bound, and we believe better bounds can be shown.

Finally, we assume that the certificate contents contain sufficient entropy to rule out brute-force inversion of the certificate hash (which is revealed to the proxy). Brute-forcing would reveal the client's identity. This requirement is easily satisfied as long as the certificate includes the client's public key. Lastly, the CA can later link a certificate it sees to a protocol execution, simply by checking a hash of the certificate against hash values produced during the blind CA registration process. We leave as an open question now to build efficient blind CAs that enjoy an unlinkability property.

VI. Design and Implementation

PAO prototypes implementation. We use Python to implement prototypes of SMTP-based PAO. The TLS library we used is tlslite, a pure Python-based open source library [45]. We modify tlslite 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 hashing.

For our SMTP-based PAO, we use a Python SMTP library named smtplib to send email [40]. 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.

We use the ABY framework [13], a state-of-the-art SMC library that provides good performance, to implement an honest-but-curious SMC for 2P-CBC for the proof-of-concept prototype. (The performance of our prototype might be reduced greatly once we replace the honest-but-curious SMC with a malicious SMC.) We implement two command-line programs; one program will be called by the client with input the TLS session key, and the other program will be running at the proxy to provide the challenge to be injected. The computation result is configured to be only given to the proxy.

Communications between client and proxy. The proxy listens on two ports, one for data forwarding (data port) and one for receiving control messages (control port). A client needs to build two connections, data connection and control connection, to the proxy. The proxy will forward the data that arrives at the data port to the corresponding destinations (and backward), but may intercept the data based on the control messages received. We define various types of control messages, and each type of message represents a given command, such as "register new user", "set forwarding address", and "start AES SMC computation". The client and the proxy interact with each other by exchanging control messages. For example, in 2P-HMAC, the client sends the internal HMAC state in a `2P-HMAC-READY` message to the proxy. Once the proxy receives this message, it knows the client is ready to run 2P-HMAC protocol. The proxy extracts the state from the message, uses it to calculate the HMAC for the challenge, and then sends the calculated result back to the client in a `2P-HMAC-RESULT` message. The control messages are protected by TLS connections. We omit detailed specification of the control protocol for the sake of brevity, but will make them available when we release the prototype source code.

Once a client connects to the data port, the proxy will send back an 8-byte random, unique user ID as a response. The client will carry this ID in the control messages so that the proxy can correlate a control connection with the appropriate client, and therefore the data connection.

An attacker might spoof the client ID in a control message to gain control of other data connections. The proxy can perform extra challenge-response style checks to defeat attempts at client ID spoofing: instead of giving the client the expected response after receiving a control message, the proxy sends back a nonce and asks the client to send the nonce in the data connection, which will be dropped by the proxy silently later. Note that we cannot correlate a control connection with a data connection simply based on IP address. If the client is using Tor, the two connections might use different exit nodes.

Proxy configuration. The server can perform active probes to fingerprint the proxy and blacklist suspicious IPs. Our strategy to defeat such attacks is to host the proxy in public clouds such as Amazon EC2 and Microsoft Azure, and frequently change IPs and the listened to ports. Note that there are numerous cloud-based email clients, meaning that a SCI proxy would not stand out due to using a cloud IP address. Putting more efforts into implementation, we can implement SCI as a plugin for popular SMTP or proxy softwares to better defeat fingerprinting.

Blind CA implementation. To create a zero-knowledge protocol for certificate generation in the blind CA service, we used the ZKBoo framework [22] to create a non-interactive proof in the random oracle model. Although this framework creates a relatively large proof, the operations required to compute the proof involve only symmetric primitives (unlike other techniques for efficient zero-knowledge which require oblivious transfer). We use SHA-256 as the hash function to sign the certificate.

We implemented ZKBoo versions of SHA-256/HMAC-SHA-256 that support inputs of any length based on the examples provided in [43], and a ZKBoo version of AES-CBC based on the code in [14]. We used the technique in [7] to implement the AES S-Box. We compute the XNOR(a, b) gate as $1 \oplus a \oplus b$, and the OR(a, b) gate as $\sim((\sim(a \& a)) \& (\sim(\text{b} \& \text{b})))$. We didn't make any optimization to the code, and we believe that with further optimizations the proof size can be reduced and the performance can be improved.

VII. Evaluation

A. Anonymous PAO

In this section we evaluate the performance of our SCI protocols and show that they work with existing services such as Gmail and two university email servers.

For all experiments we hosted the proxy on an m3.xlarge instance in the US-East region of Amazon EC2. The client was running on an Ubuntu 12.04 (64-bit) virtual machine built by VirtualBox 4.3.26, and was configured with 8 GB RAM and

4 virtual processors. We used a tool called *line_profiler* [41], which can measure the execution time for each line of code, to report on the performance of various steps in the protocols.

Latency of SCI for SMTP. We start by measuring the latency of the SCI portion of the SMTP anonymous PAO, across four different settings. To avoid causing burden on real SMTP servers, we setup our own Postfix version 2.9.6 SMTP server (with pipelining disabled) on the same EC2 instance as our proxy. We chose in this experiment to not have network latency between the proxy and server in order to maximize the relative impact of performing injection. (As we will see later proxy-to-server latency has little impact on overall performance.)

We fixed the size of the challenge and the size of the message template as 32 bytes and 512 bytes, respectively. We ran the SMTP anonymous PAO for 50 rounds, collecting timing profiles generated by *line_profiler* for the proxy and the client. We focus on session latency, the total time between initiating the protocol and completing it, as seen by the client.

The client was run from two public wireless networks at different locations (labeled as *Loc1* and *Loc2*). Then, in the best-performing wireless network, we configured the client to use Tor networks to communicate with the proxy and repeated the experiment. Finally, we ran the experiment again in a campus network (labeled as *Loc3*) with Tor being used. When Tor is being used, we let the Tor client randomly select the routers using its default algorithm, and also used the top 3 high-performance Tor routers (as reported in [44]), to build circuits. The client was instructed to stick with one of the circuits in each round by Tor control commands.

As shown in Table 8, using Tor incurs significantly high protocol overheads as one would expect, due to the high latencies of Tor networks. Using public wireless networks achieves acceptable performance. As one would expect, 2P-CBC, with its general-purpose SMC computation, accounts for the bulk of time.

The overheads introduced by SMC operations are relatively stable; in the best-performing setting, using SMC-based AES to encrypt one block of message takes approximately 0.6 s (our circuit requires AES computations for two message blocks).

As a baseline, it takes approximately 3 s and 0.3 s to send the same email in a conventional SMTP execution from the client in the settings using Tor and the settings without Tor respectively. Thus the latency overhead of SCI is about an order of magnitude in each case. Improvements to the state-of-the-art in SMC for computing AES (an active area of research [3], [13], [23]) will immediately translate into better performance for our SCI protocols and help us close this gap.

Challenge size. We measure the impact of the challenge size on protocol overhead. In the best-performing setting (*Loc2*, no Tor), we ran the same experiment as above but with challenges sizes of 2^i bytes ($i \in 5, ..., 10$). The overhead of 2P-CBC increases linearly with the challenge size. When the challenge size is 1,024 bytes, the overhead of 2P-CBC is as high as 40 seconds, which is prohibitively time consuming; on the contrary, the overhead of 2P-HMAC only has a negligible increase (by about 5 ms). This suggests that SCI is probably only feasible in practice for short challenges; our anonymous PAOs applications only require short challenges.

	Gmail	Edu1	Edu2
Normal	0.54	0.92	0.87
with PAO	1.89	2.27	2.25

TABLE 9: Average duration (in seconds) of anonymous PAO sessions and normal sessions against real SMTP services.

We further examined the timing profiles collected during this experiment in detail to find the bottleneck of the protocols. We realized that the bottleneck in 2P-HMAC is network (accounts for about 30% of the protocol overhead), while in 2P-CBC, it is CPU. The CPU time in 2P-CBC accounts for more than 80% of the protocol overhead.

Tests with real services. We tested our anonymous PAO implementation for SMTP against real services using *Loc2* without Tor. We chose Gmail and two SMTP servers at different universities as our targets. For each service, we ran the PAO protocol 10 times with a 32-byte challenge. We also measure the durations of sessions without SCI, when one sends an email directly to the service. Here the duration of a session is the total time the client spent on issuing a connection to the target service, sending an email, and closing the connection.

As shown in Table 9, on average anonymous PAO sessions will be about 1.4 seconds longer than the normal sessions; 2P-CBC contributes the bulk of the overhead.

Server obliviousness and protocol duration. The fact that a SCI SMTP session will be 1.5 – 2 seconds longer than an SMTP session in the best-performing settings might suggest that server oblviousness could be violated by simply looking at how long SMTP sessions last. Longer sessions would seemingly be indicative of SCI. But actually this alone would not be a very good detector because of the long tail of slow (conventional) SMTP sessions seen in practice. We extract and analyze the durations of 8,018 SMTP-STARTTLS sessions from a dataset of terabytes of packet-level traffic traces collected from campus networks.[5] The distribution of the SMTP durations is long-tailed. As shown in Figure 10, about 15% of the SMTP sessions analyzed requiring more than 10 s to complete. So attempting to detect SCI in such a coarse way won't work. Of course, there could be more refined detection strategies that take advantage of, for example, inter-packet timing. We plan to examine other possible traffic analysis techniques in the future.

B. Blind CA

Recall that in blind CA the prover needs to generate a zero-knowledge proof using ZKBoo. We here evaluate the feasibility of generating this proof by evaluating micro-benchmarks for its core operations.

We used a 347-byte X.509 certificate template with a 32-byte user email account in our testing. As in [22], to achieve a soundness error of roughly 2^{-80} one needs to generate at least 136 proofs. We observed the proof computation time and the proof size increase almost linearly with the number of proofs generated.

[5]The traffic was collected for a previous project for which we had obtained an IRB exemption. We only used extracted timings from that dataset for these measurements.

	2P-HMAC			2P-CBC			Total		
	Min	Max	Median	Min	Max	Median	Min	Max	Median
Loc1 (No Tor)	0.03	0.06	0.03	1.56	1.74	1.63	1.88	2.50	1.92
Loc2 (No Tor)	0.01	0.02	0.01	1.21	1.35	1.24	1.52	1.69	1.55
Loc2 (Tor, Random)	2.52	9.75	4.12	15.61	31.17	18.01	33.95	52.10	37.53
Loc3 (Tor, Random)	1.04	6.26	2.56	12.15	32.80	15.77	16.21	42.35	29.08
Loc2 (Tor, Top3)	0.37	0.90	0.49	6.84	9.18	7.60	10.01	15.08	11.53
Loc3 (Tor, Top3)	0.33	0.68	0.38	6.47	8.51	7.14	10.41	13.19	11.02

TABLE 8: Performance of SMTP anonymous PAOs across different locations, over 50 executions in each location. Shown is the time to complete the 2P-HMAC and 2P-CBC steps, as well as the total session duration (in seconds). "Random" and "Top3" refers to the two settings that use random routers and the top 3 high-performance routers to build Tor circuits respectively.

Size of a single proof. In ZKBoo, the proof size for a given computation is only decided by the number of AND/ADD gates in the corresponding arithmetic circuit of that computation. Many operations such as XOR, NOT, bit-shifting and bit-rotations do not contribute to the proof size. In our implementation, doing one round of a compression function (hashing 64 bytes of message) in SHA-256 requires 728 AND/ADD gates, and encrypting 16 bytes (1 block) of message requires 10,704 AND/ADD gates. Other computations in the proof generation do not involve AND/ADD gates. In AES, one round of the S-Box lookup operation uses 63 AND gates, and the total number of AND gates used by S-Box lookups is 10,080, which accounts for 94% of the AND/ADD gates being used. The total number of AND/ADD gates being used is 73,696. The resulting proof size is 590,824 bytes.

Proof generation and verification time. The computation time for generating one proof is about 17.9 ms (over 50 rounds of measurement); the computing time for encrypting one block in AES is about 672 us, and doing one round of compression function in SHA-256 costs about 586 us on average. Though AES uses significantly more gates than SHA-256, the computing time doesn't increase greatly. We observed that one AND/ADD computation uses about 35 ns; so, the total time for AND/ADD computations is only about 375 us in AES and 25.5 us in SHA-256. However, the AES implementation we used is byte-oriented and is using an optimized algorithm; meanwhile, the SHA-256 implementation is just based on the naive algorithm, operated on 32-bit words, and contains several timing-consuming copy operations. The verification time for the proof is about 15.3 ms on average. 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 setting).

It takes about 2.5 s and 2.1 s to generate and verify 136 proofs respectively. The total size of all the proofs together is about 80.3 M. While large, we expect that generating proofs will be an infrequent task in deployments. We also believe that further engineering effort could significantly improve performance.

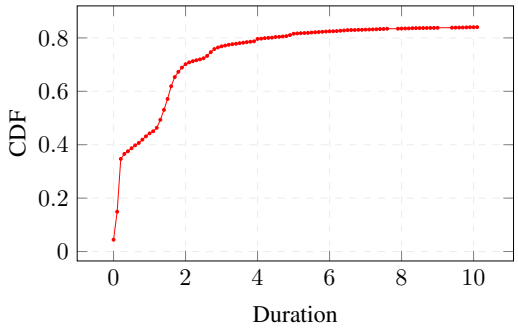


Fig. 10: CDF of duration of normal SMTP-STARTTLS sessions seen on a university campus.

VIII. Related Work

Secure multi–party computation. Secure multi-party computation (SMC) 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 [21], [49]. SCI can be seen as a special-case of SMC, but using general-purpose protocols would be prohibitively expensive and we limit their use. As such we benefit from the now long literature on making fast SMC implementations [4], [8], [13], [24].

Anonymous credentials. Anonymous credentials systems allows a user to prove to another party that she has a valid certificate issued by a given certificate authority, without revealing contents of the certificate [9], [12], [25], [38]. Some systems focus on solving the privacy issues during certificate issuing, 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 [5], [39]. 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. Our approach also requires no changes to existing SMTP services meaning it is readily deployable.

Multi-context TLS. Multi-context TLS (mcTLS) [36] is a modification to TLS that gives on-path middleboxes the ability to read or write specified *contexts* of a TLS session. A context is just a portion of the plaintext data within a session. It could be an HTTP request header, a request, or multiple requests. 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.

IX. Conclusion

We proposed secure channel injection (SCI), which allows a proxy to inject 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 showed how SCI can be used to perform anonymous proofs of account ownership (PAO), in which a user can prove ownership of some email addresses on a service, but not reveal to the verifier which one. We further build a blind CA protocol that checks email ownership and signs an X.509 certificate binding the email to a user's public key, all without ever learning the exact email or public key. Our performance evaluation demonstrates that SCI is efficient enough to use in practice and we verified that it works to perform anonymous PAOs with existing SMTP services like Gmail.

References

[1] 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.

[2] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *CRYPTO'96*, 1996.

[3] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *SP'13*, 2013.

[4] A. Ben-David, N. Nisan, and B. Pinkas, "Fairplaymp: A system for secure multi-party computation," in *CCS'08*, 2008.

[5] V. Benjumea, J. Lopez, J. A. Montenegro, and J. M. Troya, "A first approach to provide anonymity in attribute certificates," in *PKC'04*, 2004.

[6] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and G. NIST, "Keccak and the sha-3 standardization," 2013.

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

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

[9] J. Camenisch and E. Van Herreweghen, "Design and implementation of the idemix anonymous credential system," in *CCS'02*, 2002.

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

[11] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-damgård revisited: How to construct a hash function," in *CRYPTO'05*, 2005.

[12] 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 *SP'16*, 2016.

[13] D. Demmler, T. Schneider, and M. Zohner, "Aby-A framework for efficient mixed-protocol secure two-party computation." in *NDSS'15*, 2015.

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

[15] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4346.txt

[16] ——, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[17] Y. Dodis, T. Ristenpart, J. Steinberger, and S. Tessaro, "To hash or not to hash again? (In) differentiability results for H^2 and HMAC," in *CRYPTO'12*, 2012.

[18] 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 *IMC'15*. ACM, 2015.

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

[20] T. K. Frederiksen, J. B. Nielsen, and C. Orlandi, "Privacy-free garbled circuits with applications to efficient zero-knowledge," in *CRYPTO'15*, 2015.

[21] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, 2009.

[22] I. Giacomelli, J. Madsen, and C. Orlandi, "Zkboo: Faster zero-knowledge for boolean circuits," in *Usenix Security'16*, 2016.

[23] S. Gueron, Y. Lindell, A. Nof, and B. Pinkas, "Fast garbling of circuits under standard assumptions," in *CCS'15*, 2015.

[24] W. Henecka, A.-R. Sadeghi, T. Schneider, I. Wehrenberg *et al.*, "Tasty: Tool for automating secure two-party computations," in *CCS'10*, 2010.

[25] S. Hohenberger, S. Myers, R. Pass, and A. Shelat, "Anonize: A large-scale anonymous survey system," in *SP'14*, 2014.

[26] 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'16*, 2016.

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

[28] M. Jawurek, F. Kerschbaum, and C. Orlandi, "Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently," in *CCS'13*, 2013.

[29] keybase.io, "Public key crypto for everyone," https://keybase.io/, 2016.

[30] J. Klensin, "Simple Mail Transfer Protocol," RFC 5321, Internet Engineering Task Force, 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5321.txt

[31] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Internet Engineering Task Force, 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2104.txt

[32] G. Lindberg, "Anti-spam recommendations for SMTP MTAs," 1999.

[33] D. McGrew and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)," RFC 6655, Internet Engineering Task Force, 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6655.txt

[34] D. McGrew and J. Viega, "The Galois/Counter Mode of operation (GCM)." [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

[35] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "Coniks: Bringing key transparency to end users," in *Usenix Security'15*, 2015, pp. 383–398.

[36] 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 *SIGCOMM'15*. ACM, 2015.

[37] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," RFC 7539, Internet Engineering Task Force, 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7539.txt

[38] C. Paquin and G. Zaverucha, "U-prove cryptographic specification v1.1," Tech. Rep., 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/u-prove-cryptographic-specification-v1-1-revision-3/

[39] S. Park, H. Park, Y. Won, J. Lee, and S. Kent, "Traceable Anonymous Certificate," RFC 5636, Internet Engineering Task Force, 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5636.txt

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

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

[42] Sendmail, "Sendmail.com: The messaging infrastructure experts," http://www.sendmail.com, 2016.

[43] Sobuno, "Zkboo," https://github.com/Sobuno/ZKBoo, 2016.

[44] Tor community, "Tor network status," https://torstatus.blutmagie.de/, 2016.

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

[46] J. B. Ullrich, "Browser fingerprinting via SSL client hello messages," https://isc.sans.edu/forums/diary/Browser+Fingerprinting+via+SSL+Client+Hello+Messages/17210, 2013.

[47] wikipedia, "Anti-spam techniques," https://en.wikipedia.org/wiki/Anti-spam_techniques, 2016.

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

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

[50] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Transport Layer Protocol," RFC 4253, Internet Engineering Task Force, 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4253.txt