Ticket to Hide: Practical, Private Proofs of Provenance for TLS

Ryan Little
ryanlit@bu.edu
Boston University
Boston, MA, USA

Daniel S. Roche*
roche@usna.edu
United States Naval Academy
Annapolis, MD, USA

Mayank Varia
varia@bu.edu
Boston University
Boston, MA, USA

Abstract

When using Transport Layer Security (TLS), web users can connect to a server and trust that they are sending and receiving data with the intended web server. This guarantee, however, is not transferable: there is no immediate way for a client to convince an external party that a transcript or message originated from a particular server. Beginning with the DECO protocol of Zhang et al., there has been a line of work on "TLS oracles"—cryptographic protocols that allow a client to commit to, prove provenance, and disclose arbitrary properties of TLS application data to a verifier party. TLS oracles only require the server to run standard TLS, making them compatible with existing real-world web servers.

In this work we introduce Ticket to Hide, a new TLS oracle protocol for TLS 1.3. We operate in the multi-server setting, previously explored in the DiStefano protocol by Celi et al., in which the client additionally wishes to hide the identity of the server they are communicating with among a set of N publicly known servers. We leverage new features of TLS 1.3 in surprising ways to yield performance and security benefits, resulting in a protocol that is both faster and more private than previous work. Additionally, we are the first TLS oracle protocol to be compatible with post-quantum secure TLS key agreement and certificates. Our implementation, which builds on top of the Garble-then-Prove framework of Xie et al., scales to $N = 100$ servers in less than 3 seconds of end-to-end time in a WAN setting—only 3.5× the latency of a regular TLS 1.3 interaction.

1 Introduction

Transport Layer Security (TLS) is ubiquitous on the web: it is used to safeguard around 95% of web pages visited on Chrome [27] and Firefox [39]. The newest TLS 1.3 standard [53] provides forward secrecy by default, which protects prior TLS sessions even if the client or server is later coerced or compromised. In particular, an adversary cannot provably reveal TLS session data after the fact.

Against this backdrop, *TLS oracles* have been constructed to deal with an intriguing question: what if the client purposely wants to show the provenance of some data—namely, that it is copied verbatim from a TLS session with the server? Introduced in the DECO protocol by Zhang et al. [74], TLS oracles [13, 16, 17, 20, 37, 41, 57, 69] enable a client to interact with an independent verifier *during* the TLS session with the server, produce a commitment to the authenticated traffic, and convince the verifier the committed value comes from a valid interaction with the server. (The client can then prove properties of this data in zero knowledge.)

In this work, we consider TLS oracles in the *multi-server* setting explored by Santos [57] and Celi et al. [13], in which the proving client wishes to hide which one out of N possible servers it interacts with. Consider the following example statements:

- Alice has a bank account with a balance of at least $1000.
- Alice earned a grade of A in Crypto at an accredited college.
- Alice received the flu vaccine from a healthcare provider.

A multi-server TLS oracle allows Alice to prove these statements to a verifier Bob, without revealing which server was used to make the attestation. Note that verifier Bob trusts the (public set of) banks, colleges, or healthcare providers; as a result, he will be convinced if Alice logs into any one of their websites and navigates to a page that shows the desired bank balance, transcript, or health record.

While each example could be addressed by having the server run a bespoke cryptographic protocol, TLS oracles provide *flexibility* and *backward compatibility* to handle all of these examples and more without making any changes to existing web sites and servers. In other words: it is substantially easier for Alice and Bob to agree to use a TLS oracle than it is to convince all banks, colleges, or healthcare providers to implement a new cryptographic credential.

Under the hood, TLS oracles typically operate using secure two-party computation (2PC) between the prover and verifier to emulate a TLS client during the TLS handshake and application phases, a commitment to the application data & keys, and a zero knowledge (ZK) proof that the committed data comes from the interaction with the server. Early TLS oracles like DECO [74] and Garble-then-Prove [69] focused on the older TLS version 1.2 standard, which uses a simpler key derivation function and supports more 2PC-friendly modes of operation. Recent TLS oracles like Janus [37] and DiStefano [13] extend these techniques to work with the newer TLS 1.3, but add a significant performance cost as shown in Fig. 5.

1.1 This work

This work contributes the *Ticket to Hide* protocol, which for the first time implements a TLS 1.3 oracle that is **faster** than existing TLS 1.2 oracles. To achieve this goal, we show how several specific features introduced in TLS 1.3—like streamlined session resumption tickets, separate handshake vs. application keys, and the encrypted client hello (ECH) extension—can be leveraged to move work out of the 2PC and into cleartext evaluation by either the prover or verifier (see §3 for details). This has benefits beyond performance:

- Ticket to Hide is amenable to post-quantum TLS, since key agreement and signature verification in the TLS handshake are performed outside of the 2PC (see §5.7 for details).
- Ticket to Hide supports the multi-server setting, with only the 'selected' server requiring a 2PC connection.

The upshot is that Ticket to Hide runs with up to 18 servers faster than Garble-then-Prove [69] runs with one server (see Fig. 6).

We also provide a formal security analysis for Ticket to Hide. We create a TLS multi-server oracle functionality $\mathcal{F}_{\text{MSO}}$ in the UC

*The views expressed in this document are those of the authors and do not reflect the official policy or position of the U.S. Naval Academy, Department of the Navy, the Department of Defense, or the U.S. Government.

	Setting			MPC Details	
Scheme	Multi-server	Network adversary	TLS version	Boolean 2PC	ECtF
DECO [74]	✗	✓	1.2	Authenticated garbling	Paillier-based
Garble-then-Prove [69]	✗	✓	1.2	SH garbling + ZK proof	OT-based
Janus [37]	✗	✓	1.3	Dual execution	Paillier-based
Origo [20]	✗	✗	1.3	None	None
Proxying is Enough [44]	✗	✗	1.3	None	None
k-PECO [57]	✓	✗	1.2	Authenticated garbling	Unspecified
DiStefano [13]	✓	✗	1.3	Authenticated garbling	OT-based
This work	✓	✓	1.3	SH garbling + ZK proof	None

Table 1: Comparison of prior work on TLS oracles. The "Network Adversary" column indicates whether the protocol maintains privacy against an adversary with full network control (see §1.1 and §3.1). All schemes achieve malicious security, so the "Boolean 2PC" column credits schemes that use faster building blocks: no 2PC (fastest), semi-honest garbled circuits, dual execution, or authenticated garbling (slowest); see §1.2 and §2.2 for details. "ECtF" refers to a specialized MPC protocol used to convert elliptic curve point sharings to field sharings. 'None' indicates that computation is done locally, without MPC.

security framework of Canetti [10], along with an abstraction of the base TLS 1.3 protocol $\mathcal{F}_{\mathsf{TLS}}$ that focuses on the features we use in Ticket to Hide. Then, we prove that the Ticket to Hide protocol π_{TTH} UC-realizes $\mathcal{F}_{\mathsf{MSO}}$ with static security against a malicious prover or verifier. Along the way, we identify some security issues with DiStefano [13], and we show how to resolve them in Ticket to Hide.

Our contributions. Concretely, this paper makes 4 contributions.

- In §3.1 and Appendix B, we show the inherent challenges associated with TLS oracles in the multi-server setting by describing three attacks on prior protocols.
- In §4, we use the insight from these attacks to develop a UC formalism of multi-server TLS oracles $\mathcal{F}_{\mathsf{MSO}}$ (Fig. 13).
- In §5, we propose our Ticket to Hide protocol π_{TTH} that UC-realizes $\mathcal{F}_{\mathsf{MSO}}$. Our multi-server TLS 1.3 oracle protocol provides stronger security than previous work, and it supports post-quantum TLS modes with minimal overhead.
- In §6, we provide an open-source implementation of our protocol and experimentally show that it is more efficient than all prior TLS oracles—even those in the single server setting and/or that only support TLS 1.2.

Threat model. We consider $N + 2$ parties: a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and N servers $\mathcal{S}_1, \ldots, \mathcal{S}_N$. An honest prover $\mathcal{P}$ intends to prove that a commitment com contains the ciphertext and associated keys of a TLS session with its *selected* server $\mathcal{S}_{\mathsf{sel}}$. The remaining $N - 1$ servers are called *decoy* servers. Intuitively, a malicious prover seeks to deceive the verifier into thinking that some com* came from a server in $\{\mathcal{S}_1, \ldots, \mathcal{S}_N\}$ when it did not, and a malicious verifier seeks to learn additional information about the prover's decrypted communications with the servers or the identity of the real server.

More precisely, either prover $\mathcal{P}$ or verifier $\mathcal{V}$ can be malicious, and a malicious $\mathcal{V}$ can collude with one or more decoy servers (which reduces the anonymity set of the multi-server TLS oracle). Note that any larger collusion set would ruin the privacy of TLS oracles: if $\mathcal{P}$ and $\mathcal{V}$ collude then the proof is meaningless, if $\mathcal{V}$ colludes with the selected server $\mathcal{S}_{\mathsf{sel}}$ then it knows the data in com, and if $\mathcal{P}$ colludes with any server then it controls the data in com.

Malicious parties can deviate arbitrarily from the stated Ticket to Hide protocol. They also have full control of the network in order to read, delay, drop, reorder, or inject data. In particular, they can mount person-in-the-middle attacks between uncorrupted parties. We prove UC security in the static setting, in which the servers are global entities who can establish TLS sessions with anyone (not just $\mathcal{P}$ and $\mathcal{V}$). Finally, all parties are computationally bounded.

1.2 Related Work

In this section, we survey known TLS oracles and other prior work.

Early approaches. The TLSNotary protocol [24] is perhaps the first work to look at the problem of proving statements about data concealed in a TLS transcript—early versions date back to 2014. However, it requires a trusted external party to achieve privacy. Another early work, TownCrier [73], replaced this trusted party with a Trusted Execution Environment (TEE), which leaves their protocol vulnerable to attacks against TEEs [9, 15, 22, 38, 40, 64].

TLS oracles. TLS oracles have been studied extensively in the past five years, and we compare Ticket to Hide to prior works in Table 1. In this work, we focus on TLS oracles that provide security against a network adversary: that is, a malicious prover or verifier can eavesdrop on internet traffic between other parties.

DECO [74] introduced the concept of TLS oracles as well as the design template that many subsequent works (including this one) follow, described in §2.1. DECO's setting only considers a single server, and their protocol is based on TLS 1.2 (their proposed modification to support TLS 1.3 has since been shown to be insecure [13]). It uses authenticated garbling [66] for the bulk of its MPC.

Several followup works provide improvements, generalization, and modifications to DECO. The Garble-then-Prove protocol by Xie et al. [69] replaces authenticated garbling with a substantially more efficient combination of semi-honest garbled circuits and an interactive zero-knowledge proof, as we describe in §2.2. DIDO [16] extends DECO's approach to TLS 1.3, but it only works in a semi-honest setting; the follow-up work DIDO+ [17] partially addresses this issue by making components of their protocol maliciously

secure, but the overall protocol is only semi-honest secure. The Janus protocol by Lauinger et al. [37] provides support for TLS 1.3, and uses dual execution [32] (which provides malicious security with 1-bit leakage) and an honest-verifier ZK proof system [33].

TLS oracles without network visibility. There are several works that offer TLS oracles in a restricted threat model where the adversary lacks visibility into the network. We stress that this is a weaker threat model than Ticket to Hide achieves (cf. §1.1).

In the single-server setting: the works of Ernstberger et al. [20] and Luo et al. [44] are very efficient (they do not need any secure two-party computation), but they rely heavily on a direct network connection between the verifier and the server. If the prover can intercept this network traffic, then $\mathcal{P}$ can break soundness.

In the multi-server setting, the k-PECO protocol by Santos [57] (for TLS 1.2) and the DiStefano protocol by Celi et al. [13] (for TLS 1.3) rely on a direct communication channel between the prover and server. If the verifier eavesdrops on this connection, then $\mathcal{V}$ can identify the selected server (see §3.1). Additionally, DiStefano never reveals TLS handshake keys to $\mathcal{V}$ (unlike single-server protocols and Ticket to Hide), which leads to a subtle attack on soundness because $\mathcal{V}$ lacks the data to verify $\mathcal{P}$'s actions (see Appendix B).

Related crypto protocols. The key ingredient of TLS oracles—computing TLS in MPC—has been studied in other works as a standalone primitive [1] and as part of solutions to different problems like ZK middleboxes [28, 72], distributed two-factor authentication [62], private inputs to MPC and FHE [58], and producing publicly verifiable credentials and signatures over web data [46, 49]. Efficiency improvements to TLS oracles may be of use in all these problems.

Several works, beginning with DECO, have investigated optimized approaches for proving in zero-knowledge that a string can be validly parsed according to the rules of a context-free grammar [3, 45, 74]. These techniques are directly applicable to TLS oracles. In practice, many web API responses are specified in context-free grammars (e.g. JSON), and proving statements about these responses requires parsing the response in zero-knowledge.

2 Background

In this section, we provide an overview of TLS oracles, the garble-than-prove paradigm, and two features of TLS 1.3 that we use in Ticket to Hide. We defer to Appendix A: a glossary of symbols, a summary of TLS 1.3, and definitions of crypto primitives.

2.1 TLS Oracles

A TLS oracle allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that they jointly engaged in a TLS exchange with a server whose session data is contained inside a commitment com, without revealing the session data. In the multi-server setting, $\mathcal{P}$ additionally wishes to hide the server's identity among a set of agreed-upon servers $\{\mathcal{S}_1, \ldots, \mathcal{S}_N\}$. Most TLS oracle protocols have three stages: the three-party handshake, the encryption phase, and the proof phase.

Three-party handshake. In this phase, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ establish secret shares of genuine TLS application keys in an active TLS session with a server. This turns the TLS handshake—normally a two-party protocol between client and server—into a three-party protocol between the prover, verifier, and (unmodified) server. Existing TLS oracles use secure two-party computation (§A.3) for $\mathcal{P}$ and $\mathcal{V}$ to jointly emulate the client-side work of TLS.

It is crucial that neither $\mathcal{P}$ nor $\mathcal{V}$ can learn the application keys at this point. If the prover $\mathcal{P}$ learned the keys, they could encrypt an arbitrary message and claim that it was sent by the server $\mathcal{S}$, breaking soundness. If the verifier $\mathcal{V}$ learned the keys, they could decrypt all application traffic between the $\mathcal{P}$ and $\mathcal{S}$, breaking privacy.

Encryption phase. Next, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ jointly encrypt the prover's request req (such as an API request or login data) under the secret-shared TLS client application encryption key. This stage is also performed inside 2PC to avoid revealing the plaintext to $\mathcal{V}$ or leaking the full key to either party. This ciphertext is sent to the server $\mathcal{S}$, who replies with an encrypted response resp.

Proof phase. In this phase, the prover $\mathcal{P}$ provides a commitment com to the TLS application data and associated keys to $\mathcal{V}$. Then, the verifier $\mathcal{V}$ reveals all of their secrets to $\mathcal{P}$ (note that $\mathcal{V}$'s secrets were all ephemerally generated during the protocol itself, such as secret shares of key material). The prover $\mathcal{P}$ uses these secrets to decrypt the TLS application data (which $\mathcal{P}$ keeps to itself) and prove to $\mathcal{V}$ in zero knowledge that the ciphertext in commitment com is consistent with the handshake and application keys.

2.2 The Garble-then-Prove Paradigm

Introduced by Xie, Yang, Wang and Yu [69], the Garble-then-Prove paradigm is an approach to securely compute functions in the presence of malicious adversaries. They relax the standard problem of two-party computation by allowing one party P_1 to cheat—potentially modifying the function—*so long as this cheating is detectable by the other party.* Further, the other party P_2 does not have input privacy: their input is revealed even if both parties are honest.

The core idea is as follows. The parties run a standard garbled circuit protocol for function f (e.g. [43, 56, 71]), where P_1 is the garbler and P_2 is the evaluator. Denote the inputs as x_1, x_2 and the outputs as y_1, y_2. Garbled circuits already provide one-sided malicious security against the evaluator P_2. To detect a malicious P_1, Garble-then-Prove adds a few steps:

- Before garbling: P_1 commits to its input x_1.
- After garbling: P_1 commits to its output y_1, then P_2 reveals its input x_2, then P_1 proves in ZK that $(y_1, y_2) = f(x_1, x_2)$.

This technique leverages the fact that an interactive ZK proof like QuickSilver [70] is faster than (two sided) malicious 2PC protocols like authenticated garbling [66]. In TLS oracles, $\mathcal{P}$ acts as the garbler and $\mathcal{V}$ acts as the evaluator. Recall that in TLS oracles, $\mathcal{V}$ already reveals its inputs and outputs in the proof phase, making them amenable to the Garble-then-Prove technique.

2.3 Ticket-Based Resumption + Pre-Shared Keys

In TLS 1.3, a client and a server may resume a previously established connection more efficiently using a *pre-shared key* [53]. We describe this process at a high level here and give further details in §A.2.3.

To establish the pre-shared key values, the client and server first complete a full TLS 1.3 handshake. Upon establishing application encryption keys, the server sends the client an encrypted *ticket* nonce tikid, which serves as a unique identifier. Subsequently, both

client and server derive a *pre-shared key* value (psk), and they cache the tikid and psk past the lifespan of the initial TLS connection.

To resume a TLS session, the client sends the ticket tikid to the server, who retrieves the corresponding psk from its cache. The client and server use a key exchange protocol to establish a shared secret, and they use the shared secret in combination with psk to derive encryption keys for the handshake and application traffic.

Importantly, the server does *not* send a certificate or Certificate Verify message during the reconnection handshake. Instead, the server sends a value to the client that can only be computed with knowledge of psk. The client uses this value both to resume the session and to authenticate that it is communicating with the server.

We note that ticket-based resumption in TLS 1.3 fixes many of the security problems of TLS 1.2 tickets, including lack of forward secrecy and sending tickets unencrypted [31, 63]. We intentionally do not use the 0-RTT variant of session resumption for TLS 1.3 in this work, as it does not provide forward secrecy.

2.4 Encrypted Client Hello

Encrypted Client Hello (ECH) is a recent TLS 1.3 feature designed to give clients more privacy by encrypting parts of the Client Hello message under a public key belonging to the server [54]. In particular, ECH was designed to hide the Server Name Indication (SNI) extension from network adversaries. An SNI extension tells the server which domain name the client wants to connect to. This is necessary when multiple websites are hosted at the same IP address (as is the case with websites that are fronted by a hosting service or reverse proxy). The SNI extension lets the client-facing server know which website the client wishes to access, but also reveals this information to any network snooper, since it is unencrypted. In the ECH standard, the SNI extension and other parts of the Client Hello message are encrypted under the server's public key.

ECH already has substantial support, despite not yet being an official standard. It is enabled by default on browsers including Chrome and Firefox [26, 50], and it is supported by 51% of the top one million domain names [52] through providers including Cloudflare and Meta [51]. It has also received formal analysis [7].

3 Overview of Core Techniques

This section provides a high-level overview of Ticket to Hide. We show how it overcomes challenges in the multi-server setting (§3.1) and leverages features of TLS 1.3 to improve performance (§3.2).

3.1 Challenges with Multi-Server TLS Oracles

Designing a secure TLS oracle in the multi-server setting is highly non-trivial. While this setting has been explored by prior works [13, 57], we identify 3 challenges in this section to show that prior works do not achieve security with respect to our threat model (§1.1).

Security against a network adversary. Our first observation is that if the prover only connects to the selected server $\mathcal{S}_{\text{sel}}$ during the TLS oracle protocol, then a network-snooping verifier can trivially learn the identity of $\mathcal{S}_{\text{sel}}$ by observing the prover's outgoing network connection. Both DiStefano [13] and k-PECO [57] are susceptible to this attack. To preserve the anonymity of the server, the prover $\mathcal{P}$ must create cover traffic by connecting to *all* N servers.

But properly adding cover traffic is non-trivial: simply modifying prior schemes to have $\mathcal{P}$ make $N-1$ decoy TLS connections is insufficient. This is because prior works reveal to $\mathcal{V}$ parts of the TLS connection with the selected server $\mathcal{S}_{\text{sel}}$ like the server randomness and encrypted application traffic, which a network-snooping verifier can identify among the N TLS transcripts. In contrast, our Ticket to Hide protocol carefully introduces cover traffic while (a) only requiring 2PC operations for the one connection with the selected server and (b) ensuring $\mathcal{V}$ can verify that the prover acts honestly without providing information that would allow $\mathcal{V}$ to identify $\mathcal{S}_{\text{sel}}$. Empirically, we show that this cover traffic does not significantly hinder feasibility (§6).

Hiding certificate length. TLS 1.3 encrypts the server certificate, which hides its contents but not its *length* from network eavesdroppers. TLS certificates commonly have different lengths, which can be used to identify a server within an anonymity set $\{\mathcal{S}_1, \ldots, \mathcal{S}_N\}$.

In December 2024, we conducted an experiment to check the practicality of this attack on deployed TLS servers in the wild, by attempting TLS 1.3 connections to the top 10,000 domains worldwide[1]. At the time, 9263 servers completed the TLS 1.3 handshake, and we grouped them according the sequence of record lengths returned by the server from identical client hellos. Our findings indicate a strong degredation in anonymity from observing the handshake traffic sizes: More than 17% of domains were uniquely identified by the handshake length alone, while more than 50% had an anonymity set of at most 4, and even the *largest* anonymity set contained less than 1% of the total domains.

In DiStefano [13], the prover sends the encrypted certificate for $\mathcal{S}_{\text{sel}}$ to the verifier. In Ticket to Hide, the verifier connects to all N servers and obliviously passes session resumption data to the prover; recall that resumption does not use certificates (§2.3).

Validating prover behavior. Finally, in Appendix B we identify an attack on DiStefano [13] in the multi-server setting that fully breaks soundness: it allows a malicious prover to convince a verifier they obtained arbitrary data from any TLS server. The underlying issue here is that DiStefano's three-party handshake protocol hides the handshake keys from the verifier, which provides multi-server anonymity but opens an opportunity for the prover to act maliciously without being detected. We disclosed our attack to the authors of DiStefano, who subsequently proposed a fix in which the prover uses a zero-knowledge proof to convince the verifier that the handshake keys are correct (§B) [12].

Ticket to Hide takes a different approach to key derivation in which $\mathcal{V}$ holds secrets that suffice to check $\mathcal{P}$'s actions without a zero-knowledge proof, while still not revealing all handshake keys to the verifier.

3.2 Leveraging TLS 1.3 to Improve Performance

In this section, we explain how Ticket to Hide takes advantage of pre-shared keys and encrypted Client Hello messages in TLS 1.3.

Reducing ZK proof size with pre-shared keys. A TLS oracle in the multi-server setting must convince the verifier that the prover has

[1]Top 10K server names from Cloudflare Radar Domain Rankings, https://radar.cloudflare.com/domains

established a TLS connection with one of N possible servers, without revealing which one. In prior work [13, 57], this is done by bootstrapping the selected server's signature into a ring signature [55] using a ZK proof of knowledge of signature verification [14, 21].

Ticket to Hide uses an alternative approach that does not require a ring signature. The key insight is that when resuming a TLS session with a ticket, the server's certificate and signature do *not* need to be sent or checked during the resumption connection.

In Ticket to Hide, the verifier begins by obtaining N tickets and resumption secrets, one from each server. In the process of doing so, $\mathcal{V}$ checks each server's signature locally—note that anonymity is not yet an issue. The verifier sends all N tickets to the prover, withholding the resumption secrets. The prover and verifier then perform a three-party handshake protocol that reconnects to the prover's selected server using one ticket/resumption secret pair. The reconnection handshake does not require a signature check, allowing us to avoid signature verification in 2PC or a ZK proof. Moreover, this approach does not leak the length of the selected server's certificate to the verifier, preventing the attack in §3.1.

Reducing MPC overhead with encrypted Client Hello (ECH). Even though ECH is a privacy feature that reduces performance compared to standard TLS, we use ECH to improve performance for TLS oracles. Ticket to Hide leverages the fact that ECH lets the client encrypt parts of the Client Hello message: in particular, the *client randomness*, a uniformly sampled 32-byte value. As a result, $\mathcal{P}$ can hide the client randomness from a network adversary $\mathcal{V}$. (Note that we do not need the fact that ECH lets the client encrypt domain names, nor do we assume that all servers are behind the same client-facing ECH server.)

We use the hidden client randomness to create a new way to split TLS secrets between the prover and verifier in a three-party handshake. Ignoring all fixed labels and constants, TLS encryption keys are deterministically computed from three inputs: key exchange secret, hash of the TLS transcript, and *pre-shared key* (if resuming a connection using a ticket). In prior three-party handshake protocols [13, 16, 17, 28, 37, 57, 62, 69, 74], $\mathcal{P}$ and $\mathcal{V}$ obtain XOR shares of the key exchange secret while keeping the transcript hash public, and do not use a pre-shared key. Ticket to Hide partitions rather than secret-shares this data: the verifier $\mathcal{V}$ learns the key exchange and resumption secrets, and the prover $\mathcal{P}$ learns the transcript hash. Note that an eavesdropping $\mathcal{V}$ does not know the transcript since it does not know the client randomness.

This approach gives a more streamlined way to compute many TLS handshake and application keys locally rather than using 2PC, as shown in Fig. 7. The verifier computes the key exchange locally rather than using an elliptic curve to field (ECtF) share conversion protocol as in prior works (see Table 1). More than half of the TLS 1.3 key derivation can also be performed locally (see Fig. 2). With some additional optimizations described in §5.5, we can compute all TLS encryption keys while performing just 18% of the computational work of the key derivation function in MPC.

4 Definition

We model the Ticket to Hide protocol π_{TTH} and associated ideal functionality $\mathcal{F}_{\mathsf{MSO}}$ in the Universally Composable (UC) security framework [10]. UC security is well-suited to our multi-server setting with different corruption assumptions and multiple TLS connections, and it allows us to rigorously prove security even in the presence of other protocols (such as normal TLS connections) that may run concurrently with the same or different parties.

We present the ideal functionalities at a high level in this section and provide detailed listings in Appendix D. The protocol (§5) and security analysis (§5.9) come later. We remark upfront that our ideal functionalities closely follow the structure of our Ticket to Hide protocol. This is intentional: rather than modeling TLS oracles generically, our goal is to build ideal functionalities that capture the precise level of integrity and privacy of our Ticket to Hide protocol, in the presence of powerful adversaries.

4.1 Main functionality $\mathcal{F}_{\mathsf{MSO}}$

$\mathcal{F}_{\mathsf{MSO}}$ is our main functionality for a multi-server TLS oracle. The full detail is listed in Fig. 13 in the Appendix, and we provide the important insights here. $\mathcal{F}_{\mathsf{MSO}}$ connects to the network infrastructure $\mathcal{F}_{\mathsf{TCP}}$ and $\mathcal{F}_{\mathsf{TLS}}$ as global subroutines [4], and it is meant to model how a TLS oracle can operate with the help of a trusted party:

(1) Prover and verifier agree on the set of N servers.
(2) Prover sends the index sel of its selected server $\mathcal{S}_{\mathsf{sel}}$ to $\mathcal{F}_{\mathsf{MSO}}$.
(3) $\mathcal{F}_{\mathsf{MSO}}$ makes the actual TLS connections to each server, sending each request req and receiving a response resp.
(4) $\mathcal{F}_{\mathsf{MSO}}$ forwards all N responses to the prover, who decides whether they want to proceed.
(5) If so, $\mathcal{F}_{\mathsf{MSO}}$ generates a cryptographic commitment to the request/response pair for the prover's selected index.
(6) $\mathcal{F}_{\mathsf{MSO}}$ sends the commitment to the verifier, and it sends the decommitment data to the prover.

The actual functionality $\mathcal{F}_{\mathsf{MSO}}$ in Fig. 13 is more sophisticated, in order to precisely model some details in the protocol execution.

- Before beginning the "online phase", the prover and verifier must perform TLS connections to all N servers to (respectively) retrieve ECH public keys and resumption tickets.
- The TLS connections use ECH and tickets/pre-shared keys provided by the prover and verifier, respectively (see §3.2).
- The TLS connections have two rounds of messages, roughly corresponding to the TLS handshake and application data. The N TLS connections are interleaved: all N handshake rounds happen first, then all N application data rounds.
- The online phase can fail and abort (when either $\mathcal{P}$, $\mathcal{V}$, or a server does not follow the protocol). This can occur either after the first or second round of communication.

4.2 Modeling Details and Sub-Functionalities

Our UC modeling also captures some details that we discuss below: special code to handle corrupted parties, and sub-functionalities to handle server communication and crypto primitives.

Dealing with corrupted parties. As with every UC ideal functionality, $\mathcal{F}_{\mathsf{MSO}}$ includes special code to handle adversarial corruption of the malicious verifier (potentially with some servers) or the prover.

If the verifier $\mathcal{V}$ is corrupted, then the online phase proceeds as above, with a few small changes. First, in constructing the TLS messages, the verifier does not provide the ticket pre-shared key (PSK) directly, but instead reveals to the functionality some other

intermediate secrets in the TLS key schedule as they are needed—namely, the binder key, handshake secret, and main secret. Second, an adversarial verifier receives the selected request/response commitment *just prior* to the honest prover receiving the responses. This is needed for technical reasons in our Simulator and proof, and it reflects the general idea that fairness is impossible in 2PC [18] so the adversary receives an "early preview" of the output.

If the prover $\mathcal{P}$ is corrupted, then the ideal functionality does not make actual TLS connections as described above. Instead, the adversary/simulator can make these connections on its own; then, it provides $\mathcal{F}_{\mathsf{MSO}}$ with the commitment and decommitment to the purported request/response pair. The ideal functionality $\mathcal{F}_{\mathsf{MSO}}$ then checks directly whether any of the N servers actually recorded that transcript, and if so, sends the commitment to the honest verifier.

Global functionalities for server communication. We provide two ideal functionalities $\mathcal{F}_{\mathsf{TLS}}$ and $\mathcal{F}_{\mathsf{TCP}}$ for communication with the servers running TLS. These are *global* functionalities: any party, or even the environment/distinguisher $\mathcal{Z}$, can interact with them.

First, $\mathcal{F}_{\mathsf{TLS}}$, detailed in Fig. 10, is used for the parts of TLS that we do not need to customize or modify for our protocol. The GetTicket and GetECH messages represent normal TLS connections with a single client, which authentically and privately retrieve (respectively) a TLS 1.3 resumption ticket and PSK, or a TLS 1.3 ECH config (namely, a public key). While we could add another message to $\mathcal{F}_{\mathsf{TLS}}$ to perform an actual connection (i.e., sending request and response data), that would not fit the needs of our Ticket to Hide protocol, which cannot treat these connections as a "black box."

Instead, we introduce a second global functionality $\mathcal{F}_{\mathsf{TCP}}$, detailed in Fig. 9, for the session-based transport layer that underlies TLS. Both our ideal functionality *and* our actual protocol will form and send actual TLS 1.3 records over $\mathcal{F}_{\mathsf{TCP}}$ to perform the actual request/response connections. Importantly, while $\mathcal{F}_{\mathsf{TLS}}$ is private and authenticated, the underlying $\mathcal{F}_{\mathsf{TCP}}$ is not even reliable; the adversary has full control over message contents and delivery.

Finally, because the online request/response connections between the prover and server effectively bypass $\mathcal{F}_{\mathsf{TLS}}$, we need a special third message RecordReqResp to $\mathcal{F}_{\mathsf{TLS}}$, which servers will use to simply note that a given request/response plaintext pair was received and sent over $\mathcal{F}_{\mathsf{TCP}}$ using actual TLS 1.3 records. This entails no leakage whatsoever to the adversary, but is only used by the main ideal functionality $\mathcal{F}_{\mathsf{MSO}}$ to determine the ground truth of whether a purported client/server interaction actually occurred.

In other words, when the ideal $\mathcal{F}_{\mathsf{MSO}}$ sends a commitment back to the Verifier, it is certifying that this commitment contains a request/response pair which at least one server claims actually occurred. The RecordReqResp message to $\mathcal{F}_{\mathsf{TLS}}$ is simply the mechanism by which servers can make that "claim" for the ideal functionality to check.

Local sub-functionalities for cryptographic primitives. Looking ahead, our protocol π_{TTH} relies on two sub-functionalities for secure two-party computation and interactive zero knowledge proofs. These are modeled as *local* subroutines that are used exclusively within the Ticket to Hide protocol π_{TTH} (but are absent from the ideal world with $\mathcal{F}_{\mathsf{MSO}}$). First, $\mathcal{F}_{\mathsf{2PC}}$ is an ideal subroutine for maliciously secure 2PC, adapted based on the Garble-then-Prove [69] paradigm (§2.2). The garbler/prover receives its outputs immediately, but the evaluator/verifier receives its output only after revealing its own inputs to the garbler/prover. This is acceptable for TLS oracles since the verifier does not have any long-term secrets. Second, $\mathcal{F}_{\mathsf{ZK}}$ is an ideal subroutine for ZK proofs. We copy it directly from the single-session ZK functionality by Canetti et al. [11].

5 The Ticket to Hide Protocol

We are now ready to describe the full Ticket to Hide protocol. We divide our protocol into four stages: TLS pre-processing, the three-party handshake, the encryption phase, and the proof phase. We also describe several optimizations and security improvements: reducing the cost of computing HMACs in 2PC, avoiding length-leakage attacks, supporting post-quantum TLS modes, and performing all server interactions from a single client IP address. We provide a security analysis in §5.9 and Appendix F. The protocol itself is shown in Fig. 1, with a more rigorous description in Fig. 14.

We focus our attention on the `TLS_AES_128_GCM_SHA256` ciphersuite, which uses AES-128 GCM encryption and SHA256 as a hash function. This is the sole ciphersuite designated as "mandatory-to-implement" by the TLS 1.3 specification [53], and it is widely supported by real-world servers. For simplicity, we describe the protocol assuming Diffie-Hellman key exchange over an arbitrary Diffie-Hellman group (either a finite field or elliptic curve), though Ticket to Hide can use any TLS 1.3-supported key exchange mechanism, including post-quantum secure versions (§5.7).

We follow the Garble-then-Prove paradigm of Xie et al. [69] (see §2.2), allowing us to instantiate 2PC operations with semi-honest garbled circuits and later detect malicious behavior with a ZK proof. In all 2PC operations, the prover acts as the garbler and the verifier acts as the evaluator.

5.1 TLS Pre-Processing

First, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ independently perform pre-processing steps to fetch some information from the TLS servers.

The verifier $\mathcal{V}$ establishes a session resumption ticket with all N servers. To do so, they engage in a standard TLS 1.3 handshake with each server, who responds with a ticket (encrypted under the application keys) to $\mathcal{V}$ at the end of the handshake. Upon receiving a ticket, $\mathcal{V}$ terminates the connection. The verifier stores the ticket, the pre-shared key, and the binder key for each connection; see §A.2.3 for details on the TLS 1.3 ticket generation process.

The prover $\mathcal{P}$ obtains the ECH public key of each server (as we do not assume ECH keys are known *a priori*). The ECH standard [54] lays out two methods for fetching public keys:

(1) The server may include their public key in their DNS record.
(2) The client can send a client hello message to the server containing a "GREASE" extension [6].

In Ticket to Hide, $\mathcal{P}$ fetches ECH keys with a GREASE extension rather than DNS, since we make no assumptions about the security of DNS. A GREASE extension contains randomness intended to make a regular Client Hello look indistinguishable from an encrypted Client Hello to a network snooper. The server attempts to decrypt this extension with their ECH secret key. If decryption fails, indicating the extension is grease rather than an encrypted Client Hello message, they send the ECH public key to the client,

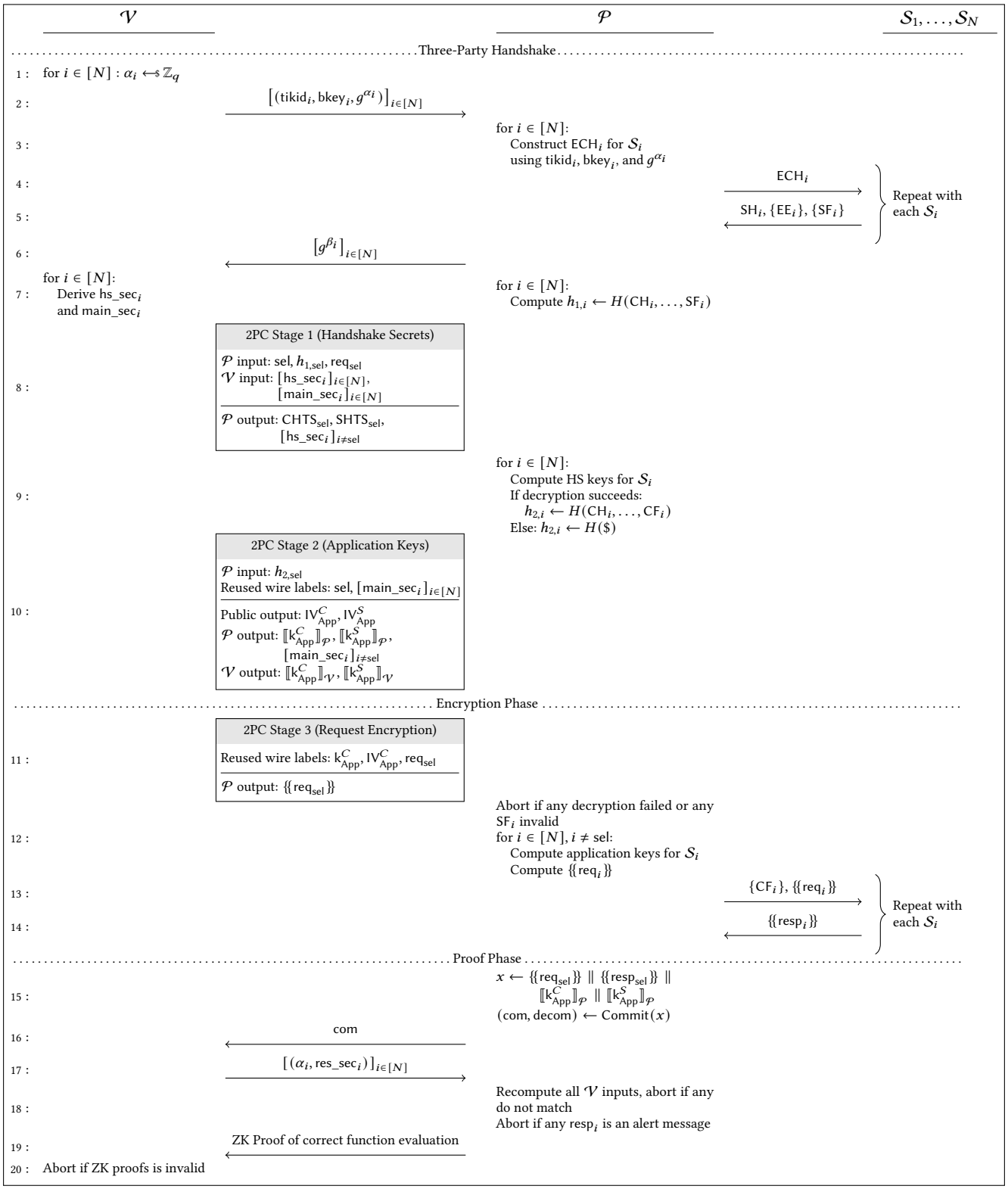


Figure 1: The Ticket to Hide protocol, informally. Values in single braces {} are encrypted under TLS handshake keys, and values in double braces {{}} are encrypted under application keys. See Figs. 14 to 16 for pseudocode from each party's perspective.

encrypted under the TLS handshake key. By fetching keys through TLS using a GREASE extension, $\mathcal{P}$ can trust the authenticity of the keys by relying on the security of TLS.

$\mathcal{P}$ and $\mathcal{V}$ do not interact with each other during pre-processing, and in theory both can perform their pre-processing well before executing the remainder of the protocol. In practice, however, parties should not run pre-processing too far in advance, since session resumption tickets expire and servers may update their ECH configs.

5.2 Three-Party Handshake

The three-party handshake is the core building block and most complicated piece of Ticket to Hide. We show the full protocol in Fig. 1 and explain the details here—with frequent references to specific steps depicted in the figure.

Key exchange. For each server $i \in [N]$, the verifier $\mathcal{V}$ samples a secret a_i and computes the key exchange share g^{α_i} to be included in the client hello message (step 1). The verifier sends each server's key exchange share to the prover, along with the ticket for that server and the binder key bkey (step 2), but retains the psk's as secrets known only to $\mathcal{V}$ and the corresponding $\mathcal{S}$.

The prover independently and uniformly samples a 32-byte cli_random string for each server. With these strings and the verifier's key exchange shares, tickets, and binder keys: the prover $\mathcal{P}$ generates an encrypted Client Hello message for each server and sends them over the network (steps 3-4). The cli_random string serves as a secret only known to $\mathcal{P}$ and the corresponding $\mathcal{S}$.

Following the TLS 1.3 spec for ticket resumption (§A.2.3), the servers respond to the ECH messages with a Server Hello message, Encrypted Extensions, and a Server Finished message (step 5). The EE and SF messages are encrypted under the handshake keys, which neither $\mathcal{P}$ nor $\mathcal{V}$ can derive yet. The prover buffers the encrypted messages for later, and sends each server's key exchange shares g^{β_i} to the verifier, for $i \in [N]$ (step 6). The verifier computes the key exchange secret for each server locally.

Jointly computing handshake secrets. We depict the key derivation process for both handshake and application keys in Fig. 2, including a breakdown of which parts are computed locally (red or blue) or with secure two-party computation (purple).

The verifier can use the key exchange secrets along with the psk secrets to locally compute the *handshake secret* (hs_sec) and *main secret* (main_sec) for each server. These are intermediate values used in key derivation. However, $\mathcal{V}$ cannot compute any further key derivation values locally. In particular, $\mathcal{V}$ cannot compute the client/server handshake traffic secrets (CHTS and SHTS) or the client/server application traffic secrets (CATS and SATS); these require the hash of the plaintext transcript, which in turn requires knowledge of the prover $\mathcal{P}$'s 32-byte client randomness value.

Instead, the prover and verifier compute the CHTS and SHTS for $\mathcal{P}$'s selected server jointly under 2PC (step 8). We henceforth refer to the connection with the selected server as the "2PC connection" and the remaining TLS connections as "decoy connections."

Since the verifier does not know which of the N resumption secrets is for the 2PC connection, they input the hs_sec for all servers. The prover inputs the index sel $\in [N]$ of their selected server, and the 2PC connection's transcript hash $h_1 = H(\mathsf{CH}, \ldots, \mathsf{SF})$. Additionally, we also have the parties provide inputs that will be used later in the protocol, for easier UC modeling. The verifier additionally inputs the main_sec for all servers, and the prover inputs the the plaintext query for the 2PC connection.

Under 2PC, $\mathcal{P}$ and $\mathcal{V}$ compute two HKDF-Expand operations [35] to compute the CHTS and SHTS respectively. These outputs are revealed in the clear to the prover, allowing them to derive the handshake keys for the 2PC connection locally. Additionally, the verifier's provided hs_sec's for all decoy connections are revealed to $\mathcal{P}$; this is done to prevent selective failure attacks by the verifier.

To selectively reveal the decoy connection handshake secrets to the prover, we introduce a gadget that we refer to as a "complement oblivious transfer": the prover inputs an index sel, and it receives all handshake secrets *except* for its selected index. Note that while it is safe for the prover to learn the hs_sec for all decoy connections, they *must not* yet learn the hs_sec of the 2PC connection, or else they could locally derive the application keys.

We implement the complement OT gadget in a garbled circuit. Our complement OT requires computing $O(N \cdot \log N)$ AND gates in a garbled circuit. We emphasize that this is the *only* MPC step in our protocol in which the size of the computation depends on N. Moreover, the concrete cost is marginal: in our implementation, for N values up to 256, each complement OT gadget requires $519 \cdot N$ AND gates (and we use two gadgets in our protocol), while the rest of the three-party handshake protocol uses ~262,000 AND gates.

Jointly computing application keys. The prover locally computes the handshake keys, decrypts the transcripts, and computes transcript hashes $h_2 = H(\mathsf{CH}, \ldots, \mathsf{CF})$ for all connections (step 9). If any transcript fails to decrypt, $\mathcal{P}$ instead hashes a uniform string, and will abort later. Then, prover $\mathcal{P}$ and verifier $\mathcal{V}$ now compute secret shares of the 2PC connection's application keys under 2PC (step 10). To do this, the prover inputs the transcript hash h_2 of the 2PC connection. The prover's previous sel input and the verifier's main_sec inputs are reused in this computation by reusing the same wire labels. Under 2PC, the parties compute the CATS and SATS value for the 2PC connection using two HKDF-Expand operations. They also derive the client and server application keys and IVs from CATS and SATS in 2PC with four additional HKDF-Expands.

Ultimately, $\mathcal{P}$ and $\mathcal{V}$ each obtain XOR shares of the client and server application keys, and learn both IVs in the clear. The 2PC also reveals all main_sec belonging to decoy connections to the prover as an output. As in the handshake key computation, this is done to stop selective failure attacks by the verifier. This is where we use the second complement OT gadget.

5.3 Encryption Phase

Next, $\mathcal{P}$ and $\mathcal{V}$ must jointly encrypt $\mathcal{P}$'s secret input under the client application key (step 11). For the TLS_AES_128_GCM_SHA256 ciphersuite, we must compute an AES-128 GCM encryption and a GCM tag. To do this, we adapt the approach of Xie et al. [69]. They compute the unauthenticated AES ciphertext in a garbled circuit and the GCM tag (with operations in $\mathsf{GF}(2^{128})$) by converting boolean shares to multiplicative shares over the field $\mathsf{GF}(2^{128})$ using Oblivious Linear Evaluation (OLE). This allows much of the tag computation to be done locally, before converting back to boolean

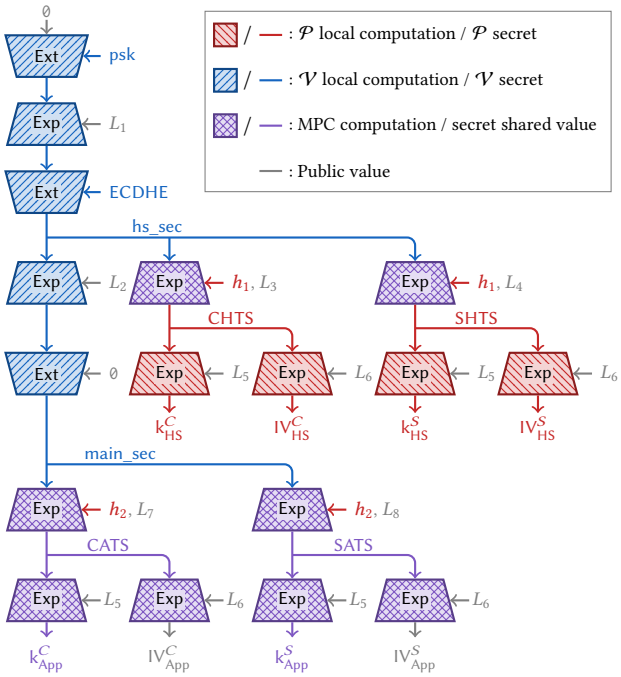


Figure 2: MPC and local computation in Ticket to Hide for TLS 1.3 key derivation. Trapezoids Exp and Ext denote HKDF-Expand and HKDF-Extract operations (respectively). HKDF internally uses HMAC: the arrow from above is the HMAC key, and the arrow from the side is the HMAC message. Also: ECDHE is the shared secret produced by key exchange, L_i values are fixed constants, and h_1 and h_2 are transcript hashes.

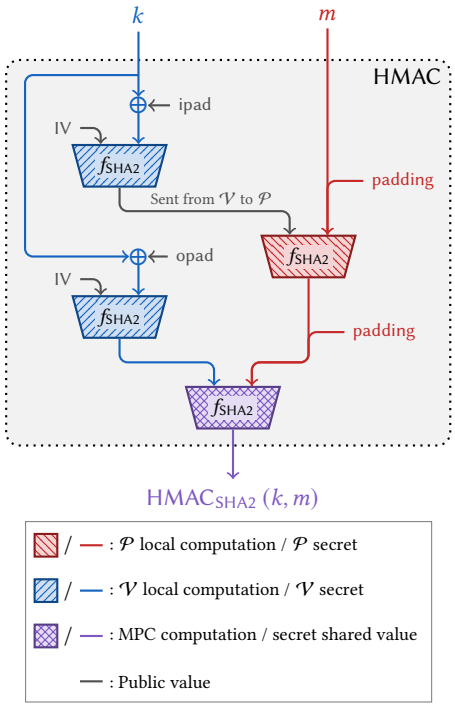


Figure 3: Our MPC optimization for computing HMAC-SHA256 in §5.5. Notation: f_{SHA2} denotes the SHA256 compression function, IV is the initial value for SHA-256, ipad and opad are inner pad and outer pad constants used in HMAC.

shares with another OLE.

There is one important difference: whereas Xie et al. [69] can reveal the ciphertext and tag to both parties, in the multi-server setting that would deanonymize the 2PC connection. Instead, in Ticket to Hide we introduce a 2PC protocol to perform the arithmetic and Boolean operations in a modified way such that $\mathcal{V}$ cannot learn the outputs; see Appendix C for details.

Additionally, the prover computes encrypted application data for all decoy servers (step 12). This is straightforward since the prover knows all decoy keys. The prover also computes CF messages for all servers. The prover aborts the protocol if any of the TLS handshakes failed to decrypt in step 9, or if any of the SF messages were invalid. Otherwise, the prover sends all CF messages and request ciphertexts to the respective servers simultaneously, and $\mathcal{P}$ receives all server response ciphertexts (steps 13-14).

5.4 Proof Phase

In this phase, the prover $\mathcal{P}$ commits to four values: the request req and response resp of the 2PC connection, and their key shares of the client and server application encryption keys (step 15-16). Then, the verifier $\mathcal{V}$ reveals its secrets to the prover: the α_i and res_sec_i values for each connection (step 17). The prover locally re-computes all functions computed under 2PC to validate the verifier's inputs, and aborts the protocol if any values do not match (step 18). The prover also decrypts all response ciphertexts, and aborts if any are an alert message (as would be sent if the server received an invalid request ciphertext).

Next, the prover constructs a ZK proof that all outputs are consistent with the inputs (step 19), as per the Garble-then-Prove paradigm (§2.2). Our protocol uses an interactive proof, following Xie et al. [69]. The verifier checks this proof, after which it is convinced that the committed data corresponds to a valid response/request interaction with one of the N servers. This completes the Ticket to Hide protocol.

We remark that if the prover wishes to prove some predicate about the committed data, $\mathcal{P}$ can do so within the aforementioned ZK proof or in a subsequent proof. As with prior TLS oracles, we do not consider such proofs within Ticket to Hide since the predicates are application-specific, though they are straightforward to add. There is one important caveat: if the prover's statement depends on the identity of the selected server (e.g., if every website formats their response differently, requiring a different predicate) then $\mathcal{P}$ must prove a disjunctive statement over the set of possible predicates.

5.5 HMAC Optimizations

The most expensive piece of our protocol is the 2PC computation of HKDF-Expand operations for key derivation. This section describes optimizations to reduce the size of this 2PC computation by more than 60%. When using the `TLS_AES_128_GCM_SHA256` ciphersuite, the HKDF-Expand function requires one HMAC operation [35],

which in turn makes four calls to the SHA256 compression function. By inlining the Merkle-Damgård structure of SHA256, we can express the $\mathsf{HMAC}_{\mathsf{SHA2}}(k, m)$ computation as:

$$f_{\mathsf{SHA2}}\left(f_{\mathsf{SHA2}}(\mathsf{IV}, k \oplus \mathrm{opad}), f_{\mathsf{SHA2}}(f_{\mathsf{SHA2}}(\mathsf{IV}, k \oplus \mathrm{ipad}), m)\right).$$

Here, f_{SHA2} denotes the compression function of SHA256, and IV, opad, and ipad are constant strings (cf. [30, 35]). Naïvely computing this function in a garbled circuit requires around 90,000 AND gates using a standard SHA256 circuit [61].

Xie et al. [69] introduced an optimization for securely computing HMACs in 2PC when the string m is public and the key k is secret-shared between two parties. Their optimization allows a single HMAC to be computed with three 2PC evaluations of f_{SHA2}, and each subsequent HMAC under the same key can be computed with only one additional 2PC evaluation of f_{SHA2}.

This optimization can be used for some, but not all, HKDF-Expands in our protocol. In each Exp box of Fig. 2, the arrow from above denotes the HMAC key, and the arrow from the side denotes the HMAC message. The four HKDF-Expands used to compute the final TLS application keys and IVs have a secret-shared HMAC key and public message string, and are therefore suitable for Xie at al [69]'s optimization (allowing those four HKDF-Expands to be computed with eight total 2PC evaluations of SHA256).

The remaining four HKDF-Expand operations work differently: the HMAC key k is a secret known to $\mathcal{V}$, and the message string m is a secret known to $\mathcal{P}$. We propose a different optimization for these operations: it requires a single 2PC evaluation of f_{SHA2} for each HKDF-Expand, without any amortization. Our optimization is depicted in Fig. 3, and we describe it here. First, the verifier $\mathcal{V}$ (who knows k) computes $h_a = f_{\mathsf{SHA2}}(\mathsf{IV}, k \oplus \mathrm{ipad})$ and $h_b = f_{\mathsf{SHA2}}(\mathsf{IV}, k \oplus \mathrm{opad})$ locally. Second, $\mathcal{V}$ sends h_a to $\mathcal{P}$; when modeling f_{SHA2} as a random oracle, h_a reveals nothing about k. Third, the prover $\mathcal{P}$ (who knows m) computes $h_c = f_{\mathsf{SHA2}}(h_a, m)$ locally. Finally, $\mathcal{P}$ and $\mathcal{V}$ jointly compute $\mathsf{HMAC}_{\mathsf{SHA2}}(k, m) = f_{\mathsf{SHA2}}(h_b, h_c)$ in 2PC.

By using our optimization and Xie et al [69]'s, we reduce the total number of 2PC SHA256 calls in our protocol from 32 down to 12. This reduces the number of AND gates computed in garbled circuits during the three-party handshake from 694,000 to 263,000.

5.6 Preventing Length-Leakage Attacks

The above protocol ensures that a malicious verifier cannot decipher any TLS-encrypted data in the connections with the selected server $\mathcal{S}_{\mathsf{sel}}$ or the decoy servers. However, $\mathcal{V}$ might still be able to use the *lengths* of the encrypted messages to identify the selected server. Ticket to Hide employs two methods to stop length leakage.

The prover's request ciphertexts must all be the same length, since $\mathcal{V}$ learns the length of the request to $\mathcal{S}_{\mathsf{sel}}$ during the 2PC encryption. To achieve this, $\mathcal{P}$ pads all request plaintexts to a fixed length, using TLS 1.3's support for arbitrarily padded plaintexts [53].

Hiding the lengths of the server's response requires a different approach, since we cannot require the (unmodified) servers to use padding. Instead, the prover $\mathcal{P}$ can use an HTTP range request [23] to specify the exact number of bytes they wish to receive from the server. The prover can send a (padded) HTTP range request for a large resource to each decoy server, requesting the number of bytes that would be served in a successful login response. To assess the feasibility of this approach, we tested 20 major international bank websites and found that 14 of them accept HTTP range requests.

We remark that a timing attack could be possible based on the server's ease of response to range requests compared to other requests. While out of scope of our threat model and security analysis, side channels are an interesting topic for future work and again highlight the challenging nature of the multi-server setting (§3.1).

5.7 Post-Quantum Support

The only officially standardized key agreement protocols in TLS 1.3—Diffie-Hellman over finite fields and elliptic curves—are vulnerable to attacks by quantum computers [59]. This has motivated a push for *post-quantum* key exchange for TLS 1.3 [2]. The main standard for this is an IETF draft [36] that takes a hybrid approach, using CRYSTALS-Kyber KEM [8] in combination with elliptic curve Diffie-Hellman. This has gained widespread adoption in recent years: in October 2025, Cloudflare reported that over half of their clients' TLS 1.3 connections used post-quantum key exchange [67].

Ticket to Hide can be easily adapted to support post-quantum key exchange. In our protocol, all key exchange computations are done outside of 2PC or ZK proofs, allowing us to modularly swap out the key exchange protocol with minimal overhead. We are the first TLS oracle with this property, aside from works that operate in the less secure proxy model [20, 44]. By contrast, adding post-quantum support to other TLS oracles protocols would require running CRYSTALS-Kyber KEM in MPC, adding significant performance costs and implementation complexity.

Separately, there has been a growing interest in post-quantum secure replacements for the signature schemes used in TLS 1.3 certificates [2, 47, 60]. While these proposals do not yet have significant adoption [68], we observe that our protocol can be extended to support post-quantum signatures too, since signature verification is done locally. We are the first multi-server TLS oracle with this property: DiStefano [13] and k-PECO [57] both compute signature verification in a ZK proof.

We note that Encrypted Client Hello does not yet support any post-quantum encryption schemes [5]. However, should a post-quantum encryption scheme for ECH be deployed, Ticket to Hide could support it with minimal overhead since Encrypted Client Hello messages are also computed outside of 2PC and ZK proofs.

5.8 Unifying the Client IP Address

In our three-party handshake protocol, $\mathcal{P}$ acts as the liaison with the servers, sending and receiving all TLS messages over the network. One quirk of this approach is that the tickets generated during TLS pre-processing are obtained from $\mathcal{V}$'s IP address, but the resumption comes from $\mathcal{P}$'s IP address. In theory, a TLS server could reject the resumption connection due to the IP mismatch.

Should this be an issue, one can mitigate the IP mismatch by proxying $\mathcal{V}$'s pre-processing connection through $\mathcal{P}$, so that all TLS messages originate from $\mathcal{P}$'s IP address. Because TLS is secure against person-in-the-middle attacks: $\mathcal{P}$ cannot learn the tickets, and $\mathcal{V}$ will detect if $\mathcal{P}$ modifies the communication in transit.

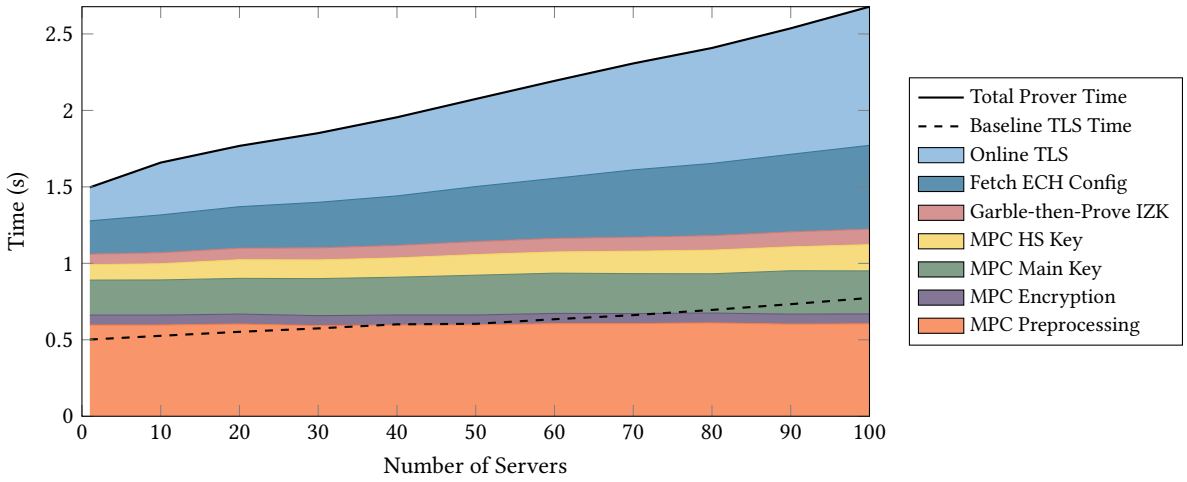


Figure 4: Total prover end-to-end running time, over WAN. Prover plaintext is 128 bytes. Each measurement is the mean of 10 samples.

5.9 Security Analysis

The privacy and integrity of our protocol are guaranteed by the following theorem.

Theorem 5.1. *Based on the cryptographic assumptions listed in Appendix A.2.1 (which are commonly used in TLS itself), The Ticket to Hide protocol π_{TTH} (Fig. 1 and Figs. 14 to 16) along with subroutines $\mathcal{F}_{\mathsf{2PC}}$ and $\mathcal{F}_{\mathsf{ZK}}$ UC-realizes functionality $\mathcal{F}_{\mathsf{MSO}}$ (Fig. 13) in the presence of global functionalities $\mathcal{F}_{\mathsf{TCP}}$ and $\mathcal{F}_{\mathsf{TLS}}$ with static corruptions.*

We provide here a high-level sketch of the proof, with details deferred to Appendix F. The analysis depends on which party/parties have been statically corrupted. If neither $\mathcal{P}$ nor $\mathcal{V}$ are corrupt then it is straightforward to see that π_{TTH} correctly provides the same outputs as $\mathcal{F}_{\mathsf{MSO}}$ and that the view of π_{TTH} is easily simulatable. On the other hand, if both $\mathcal{P}$ and $\mathcal{V}$ are corrupt then there is nobody honest left to protect. We examine the remaining cases below.

In the case of a corrupted prover $\mathcal{P}^*$, the simulator Sim takes on the role of the honest verifier $\mathcal{V}$. By contrast, the simulator Sim cannot take on the role of the server since it is a global functionality; in other words, Sim does not know the long-term secret keys of real web servers. The simulator Sim acts as follows:

- $\mathcal{F}_{\mathsf{MSO}}$ leaks the tickets tikid and pre-shared keys psk to Sim.
- Sim (acting as $\mathcal{V}$) interacts with $\mathcal{P}^*$ in π_{TTH}. Eventually, Sim receives a commitment com, and it extracts decom.
- Sim checks consistency of $\mathcal{P}^*$'s actions with the committed data, and that $\mathcal{P}^*$ acted honestly in $\mathcal{F}_{\mathsf{2PC}}$. If so, Sim sends (com, decom) to $\mathcal{F}_{\mathsf{MSO}}$; otherwise, $\mathcal{F}_{\mathsf{MSO}}$ and π_{TTH} abort.

In the case of a corrupted verifier $\mathcal{V}^*$, the simulator Sim takes on the role of the honest prover $\mathcal{P}$. The simulator Sim acts as follows:

- During pre-processing and the handshake: Sim passively observes $\mathcal{V}^*$. (If $\mathcal{V}^*$ is cheating, Sim does not yet know.)
- After the handshake: Sim (acting as $\mathcal{P}$) extracts the handshake secrets and main secrets from $\mathcal{F}_{\mathsf{2PC}}$, and passes them to $\mathcal{F}_{\mathsf{MSO}}$. Note that $\mathcal{F}_{\mathsf{MSO}}$ aborts if and only if π_{TTH} does.
- After the encryption phase: Sim extracts the message in the prover's commitment and checks it for consistency with $\mathcal{V}^*$'s earlier messages. Again, $\mathcal{F}_{\mathsf{MSO}}$ aborts iff π_{TTH} does.

If a corrupt $\mathcal{V}^*$ colludes with some servers, then the only difference is that some instances of $\mathcal{F}_{\mathsf{TLS}}$ are replaced with interactions with adversarially-controlled servers. This difference is irrelevant in all cases: if the server acts honestly then it is identical to $\mathcal{F}_{\mathsf{TLS}}$, and if it deviates from the protocol then Sim detects this in the consistency checks and aborts; once again, $\mathcal{F}_{\mathsf{MSO}}$ aborts iff π_{TTH} does.

6 Experimental Results

We provide an open-sourced implementation[2] of Ticket to Hide, comprising 7000 novel lines of Python and 2000 lines of C++. Our Python code contains a standalone TLS 1.3 implementation that supports ECH and tickets, as well as high-level networking logic for our protocol. We implement the 2PC of our protocol in C++ using the EMP Toolkit library[65]. Our C++ code is forked from an open sourced implementation of Garble-then-Prove authored by Primus Labs[3]. Under the hood, our protocol uses half-gate garbled circuits [71] for 2PC, and QuickSilver [70] for interactive ZK proofs.

6.1 End-to-End Performance Evaluation

To assess the feasibility and scalability of running Ticket-to-Hide in the real world, we benchmarked our implementation in a WAN setting with up to $N = 100$ servers. We used three AWS EC2 t2.medium instances with 2 vCPU cores and 4GB RAM: one for the prover (hosted in California), one for the verifier (hosted in Oregon), and one to run all N servers on separate ports (hosted in Virginia). We measured the bandwidth between each machine as 484.8 Mbits/second between $\mathcal{P}$ and $\mathcal{V}$, 191.2 Mbits/second between $\mathcal{P}$ and $\mathcal{S}$, and 92.54 Mbits/second between $\mathcal{V}$ and $\mathcal{S}$.

Fig. 4 presents end-to-end timing results for the prover. We test our protocol with a 128-byte prover plaintext, which is large enough to encrypt a username and password, for instance. Our

[2]https://github.com/ryanjlittle/ticket-to-hide
[3]https://github.com/primus-labs/otls

results show that Ticket to Hide can scale to up to 100 servers while running in under 3 seconds, including all preprocessing. We also measure the running time of each individual MPC components. The measurements of these reflect the total time: including garbling, sending the garbled circuit, and evaluation. The MPC preprocessing region consists mainly of setup for the interactive ZK proof, and may be computed before the live TLS connection.

The end-to-end run time of the verifier is similar to the prover time. The main difference between the two is the TLS preprocessing, which consists of the prover fetching the ECH config from each server, and the verifier fetching a ticket from each server. These two steps take a similar amount of time, as both consist of performing a TLS handshake with each server.

We also measure the "baseline" time to send and receive queries to all N servers via regular TLS 1.3 (without a TLS oracle). We find that for up to 100 servers, running Ticket to Hide is only 3-3.5 times slower than the baseline time of running standard TLS.

6.2 Benchmarking Against Prior Work

Fig. 5 compares the MPC elements of our protocol with the prior works of DECO [74], Garble-then-Prove [69], DiStefano [13], and Janus [37]. We find that Ticket to Hide outperforms all prior schemes in terms of run time, communication, and number of total AND gates. For this comparison, we restrict our scheme to the single-server setting by choosing $N = 1$. We note that the DiStefano protocol has a variant which supports the multi-server setting, but their implementation supports only the single-server setting. Our implementation is the first full-fledged TLS oracle implementation in the multi-server setting.

For each protocol, we used the authors' implementation code (when available) to measure the total run time and communication of all MPC computations between the prover and verifier, without any server communication or local TLS computations. This approach avoids measuring protocol-independent factors like network delay or the performance of an underlying TLS implementation. Each experiment was run locally on an AWS EC2 t2.medium instance with two vCPU cores and 4GB RAM. All prover and verifier processes were run single-threaded. We were unable to benchmark DECO [74], as the authors do not provide an open-source implementation.

We remark that the run-time and communication of each work is heavily dependent on the MPC protocol used to perform Boolean 2PC. For instance, DiStefano uses authenticated garbling, which has significantly higher communication overhead than the dual-execution technique used by Janus, or the Garble-then-Prove paradigm used by Garble-then-Prove and Ticket to Hide.

In Fig. 5, we also compare the total number of AND gates computed in garbled circuits for each scheme, for a full protocol execution with 128-byte prover plaintext. This gives an implementation-agnostic comparison of the MPC cost of each protocol. This measurement does not account for the cost of arithmetic MPC (all protocols we compare against, including ours, use some arithmetic MPC alongside a Boolean garbled circuit scheme). However, in practice, the total MPC cost of all schemes is dominated by the garbled circuit cost. For DECO [74], we calculate the total number of AND gates in their protocol with a 128-byte plaintext by taking their

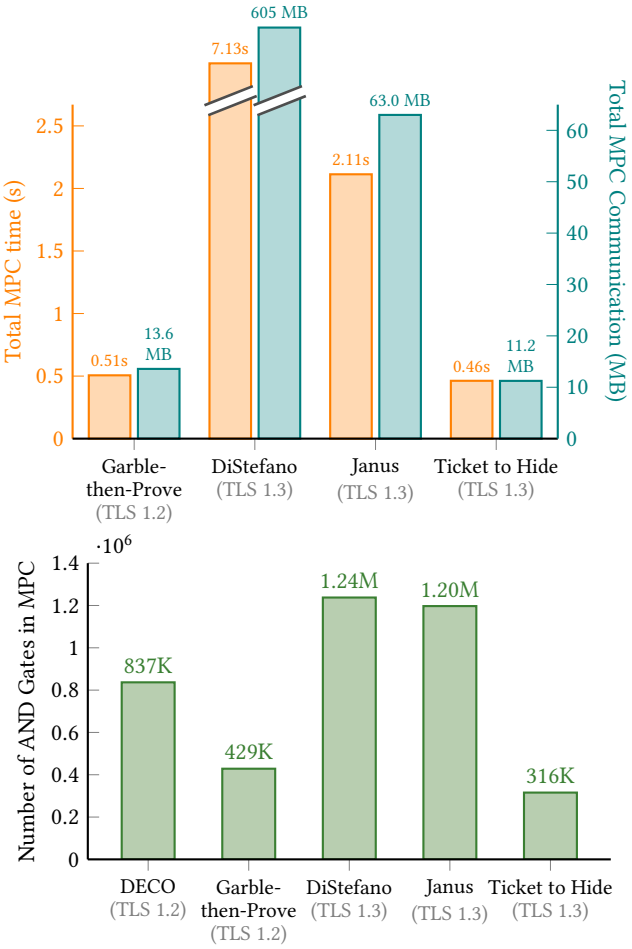


Figure 5: Comparison of MPC operations with prior TLS oracles in the single-server setting. Measurements are for a full protocol execution with 128-byte prover plaintext. Experiments run over a local network.

reported number of 779,213 AND gates for their handshake circuit and adding the number of AND gates (57,600) required to encrypt a 128-byte payload under AES-128 GCM using a standard circuit. We remark that the HMAC optimizations used by Garble-then-Prove and Ticket to Hide significantly reduces the number of AND gates over the remaining schemes, who do not use such optimizations.

Finally, Fig. 6 shows a comparison between our protocol and the scheme of Xie et al. [69]; the figure shows a variable number of servers for Ticket to Hide, whereas Garble-then-Prove [69] only supports a single server.

7 Conclusion

In this paper we present Ticket to Hide: a TLS oracle protocol that produces commitments to authenticated, encrypted data from a TLS 1.3 server, enabling web clients to produce flexible, privacy-preserving credential schemes that are backward compatible with existing web servers. We show that TLS 1.3 oracles do not need to

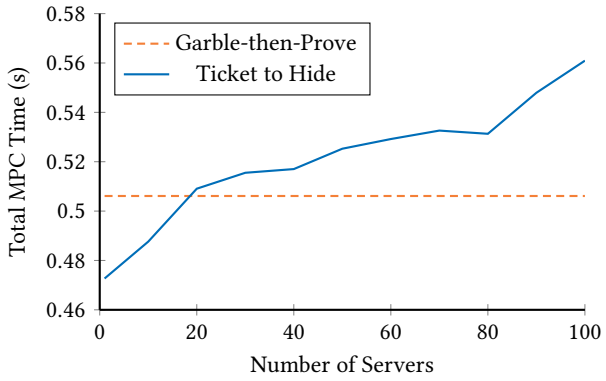


Figure 6: MPC comparison of Ticket to Hide against Garble-then-Prove [69], the fastest previous TLS oracle protocol. Garble-then-prove only supports a single server, so this comparison is to show the scalability of our protocol. Experiments were run over a local network using a 128-byte prover plaintext.

be slower than their TLS 1.2 counterparts. New features of TLS 1.3, including ticket resumption and encrypted Client Hello, have surprising benefits for TLS oracles that allow for numerous optimizations. Additionally, we further the study of the multi-server setting by identifying attacks on prior schemes, proposing a more comprehensive security definition in the UC framework, and ultimately building a protocol satisfying our definition. Our implementation of Ticket to Hide demonstrates the feasibility and scalability of our protocol: A full execution with 100 servers takes only 2.68 seconds on modest hardware in the WAN setting. These results establish TLS oracles as a promising primitive for real-world deployment.

Acknowledgments

The authors gratefully acknowledge the authors of the DiStefano paper [13] for their helpful discussion about their protocol and comparison to our work, including their feedback on Appendix B. This work was performed in part while D. Roche was graciously hosted by the Faculty of Computing & Data Sciences at Boston University. The work of R. Little and M. Varia was supported by the National Science Foundation under Grant No. 2209194 and by the DARPA SIEVE program under Agreement No. HR00112020021. The work of D. Roche was supported in part by the National Science Foundation under Grant No. 2217597.

AI tool use. All text and intellectual contributions in this paper were produced solely by the authors. In the process of this work, we used generative AI tools (ChatGPT GPT-4o and GPT-5 mini) for three applications: (1) for overall proofreading, (2) to help create source code for TikZ figures, (3) as a manual while writing source code for the implementation. All contents of this paper, including supplemental materials, are an accurate representation of our work and are correct to the best of our knowledge.

References

[1] Damiano Abram, Ivan Damgård, Peter Scholl, and Sven Trieflinger. Oblivious TLS via multi-party computation. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 51–74. Springer, Cham, May 2021.

[2] Nouri Alnahawi, Johannes Müller, Jan Oupický, and Alexander Wiesmaier. A comprehensive survey on post-quantum TLS. *CiC*, 1(2):6, 2024.

[3] Sebastian Angel, Sofía Celi, Elizabeth Margolin, Pratyush Mishra, Martin Sander, and Jess Woods. Coral: Fast succinct non-interactive zero-knowledge CFG proofs. Cryptology ePrint Archive, Report 2025/1420, 2025.

[4] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 1–30. Springer, Cham, November 2020.

[5] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. RFC 9180, February 2022.

[6] David Benjamin. Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility. RFC 8701, January 2020.

[7] Karthikeyan Bhargavan, Vincent Cheval, and Christopher A. Wood. A symbolic analysis of privacy for TLS 1.3 with encrypted client hello. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 365–379. ACM Press, November 2022.

[8] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy*, pages 353–367. IEEE Computer Society Press, April 2018.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*. USENIX Association, 2017.

[10] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[11] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.

[12] Sofía Celi, Alex Davidson, Hamed Haddadi, Gonçalo Pestana, and Joe Rowell. DiStefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more. Cryptology ePrint Archive, Report 2023/1063, 2023.

[13] Sofía Celi, Alex Davidson, Hamed Haddadi, Gonçalo Pestana, and Joe Rowell. DiStefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more. In *NDSS 2025*. The Internet Society, February 2025.

[14] Sofía Celi, Shai Levin, and Joe Rowell. CDLS: Proving knowledge of committed discrete logarithms with soundness. In Serge Vaudenay and Christophe Petit, editors, *AFRICACRYPT 24*, volume 14861 of *LNCS*, pages 69–93. Springer, Cham, July 2024.

[15] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In *2020 IEEE Symposium on Security and Privacy*, pages 1416–1432. IEEE Computer Society Press, May 2020.

[16] Kwan Yin Chan, Handong Cui, and Tsz Hon Yuen. DIDO: Data provenance from restricted TLS 1.3 websites. ISPEC 2023, 2023.

[17] Kwan Yin Chan, Handong Cui, Tsz Hon Yuen, and Siu Ming Yiu. DIDO+: Data provenance from restricted TLS 1.3 websites with selective disclosure. TDSC 2025, 2025.

[18] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*, pages 364–369. ACM Press, May 1986.

[19] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 155–186. Springer, Cham, August 2018.

[20] Jens Ernstberger, Jan Lauinger, Yinnan Wu, Arthur Gervais, and Sebastian Steinhorst. ORIGO: Proving provenance of sensitive data with constant communication. *PoPETs*, 2025(2):416–433, April 2025.

[21] Armando Faz-Hernández, Watson Ladd, and Deepak Maram. ZKAttest: Ring and group signatures for existing ECDSA keys. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 68–83. Springer, Cham, September / October 2022.

[22] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Comput. Surv.*, 54(6):126:1–126:36, 2022.

[23] Roy T. Fielding, Yves Lafon, and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Range Requests. RFC 7233, June 2014.

[24] Ethereum Foundation. Tlsnotary - a mechanism for independently audited https sessions. Retrieved January 2026 from https://tlsnotary.org/TLSNotary.pdf, September 2014.

[25] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[26] Google. Feature: TLS encrypted client hello (ECH). Retrieved January 2026 from https://chromestatus.com/feature/6196703843581952, 2023.

[27] Google. HTTPS encryption on the web. https://transparencyreport.google.com/https/overview, 2026.

[28] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 4255–4272. USENIX Association, August 2022.

[29] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 66–97. Springer, Cham, August 2017.

[30] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, May 2011.

[31] Sven Hebrok, Simon Nachtigall, Marcel Maehren, Nurullah Erinola, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. We really need to talk about session tickets: A large-scale analysis of cryptographic dangers with TLS session tickets. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 4877–4894. USENIX Association, August 2023.

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

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

[34] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[35] Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[36] Kris Kwiatkowski, Panos Kampanakis, Bas Westerbaan, and Douglas Stebila. Post-quantum hybrid ECDHE-MLKEM Key Agreement for TLSv1.3. Internet-Draft draft-ietf-tls-ecdhe-mlkem-01, Internet Engineering Task Force, September 2025. Work in Progress.

[37] Jan Lauinger, Jens Ernstberger, Andreas Finkenzeller, and Sebastian Steinhorst. Janus: Fast privacy-preserving data provenance for TLS. *PoPETs*, 2025(1):511–530, January 2025.

[38] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 487–504. USENIX Association, August 2020.

[39] Let's Encrypt. Let's Encrypt stats: Percentage of web pages loaded by Firefox using HTTPS. https://letsencrypt.org/stats/#percent-pageloads, 2026.

[40] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 717–732. USENIX Association, August 2021.

[41] Yong Lin, Peng Jiang, Fuchun Guo, and Liehuang Zhu. CrossAAD: Cross-chain abnormal account detection. In Tianqing Zhu and Yannan Li, editors, *ACISP 24, Part III*, volume 14897 of *LNCS*, pages 84–104. Springer, Singapore, July 2024.

[42] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 277–346, 2017.

[43] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. BitGC: Garbled circuits with 1 bit per gate. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part VII*, volume 15607 of *LNCS*, pages 437–466. Springer, Cham, May 2025.

[44] Zhongtang Luo, Yanxue Jia, Yaobin Shen, and Aniket Kate. Proxying is enough: Security of proxying in TLS oracles and AEAD context unforgeability. In *AFT*, volume 354 of *LIPIcs*, pages 4:1–4:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[45] Harjasleen Malvai, Gregory Neven, Andrew Miller, and Siam Hussain. Practical proofs of parsing for context-free grammars. Cryptology ePrint Archive, Report 2024/562, 2024.

[46] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. CanDID: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *2021 IEEE Symposium on Security and Privacy*, pages 1348–1366. IEEE Computer Society Press, May 2021.

[47] Jake Massimo, Panos Kampanakis, Sean Turner, and Bas Westerbaan. Internet X.509 Public Key Infrastructure - Algorithm Identifiers for the Module-Lattice-Based Digital Signature Algorithm (ML-DSA). Internet-Draft draft-ietf-lamps-dilithium-certificates-13, Internet Engineering Task Force, September 2025. Work in Progress.

[48] David McGrew and John Viega. The Galois/counter mode of operation (GCM). Retrieved January 2026 from https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf, 2004.

[49] Pierpaolo Della Monica, Ivan Visconti, Andrea Vitaletti, and Marco Zecchini. ACTS: Attestations of contents in tls sessions. In *NDSS*, 2026.

[50] Mozilla. Encrypted client hello (ECH) - frequently asked questions. Retrieved January 2026 from https://support.mozilla.org/en-US/kb/faq-encrypted-client-hello, 2024.

[51] Jonas Mücke, Konstantin Gasser, Thomas C. Schmidt, and Matthias Wählisch. Towards a complete view of encrypted client hello deployments. In *Proceedings of the ACM SIGCOMM 2025 Posters and Demos*, ACM SIGCOMM Posters and Demos '25, page 22–24. Association for Computing Machinery, 2025.

[52] Niklas Niere, Felix Lange, Nico Heitmann, and Juraj Somorovsky. Encrypted client hello (ECH) in censorship circumvention. *FOCI*, 2025(2):64–73, 2025.

[53] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[54] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni-25, Internet Engineering Task Force, March 2025. Work in Progress.

[55] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 552–565. Springer, Berlin, Heidelberg, December 2001.

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

[57] Manuel B. Santos. PECO: methods to enhance the privacy of DECO protocol. Cryptology ePrint Archive, Report 2022/1774, 2022.

[58] Manuel B. Santos, Dimitris Mouris, Xiang Xie, Miguel de Vega, and Andrei Lapets. TLShare: Private authenticated MPC and FHE inputs over TLS. Cryptology ePrint Archive, Report 2025/1434, 2025.

[59] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134. IEEE Computer Society Press, November 1994.

[60] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. Post-quantum authentication in TLS 1.3: A performance study. In *NDSS 2020*. The Internet Society, February 2020.

[61] Nigel Smart. Bristol fashion mpc circuits. Retrieved January 2026 from https://nigelsmart.github.io/MPC-Circuits.

[62] Sijun Tan, Weikeng Chen, Ryan Deng, and Raluca Ada Popa. MPCAuth: Multifactor authentication for distributed-trust systems. In *2023 IEEE Symposium on Security and Privacy*, pages 829–847. IEEE Computer Society Press, May 2023.

[63] Filippo Valsorda. We need to talk about session tickets. Retrieved January 2026 from https://words.filippo.io/we-need-to-talk-about-session-tickets, 2017.

[64] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 991–1008. USENIX Association, August 2018.

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

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

[67] Bas Westerbaan. State of the post-quantum internet in 2025. Retrieved January 2026 from https://blog.cloudflare.com/pq-2025/, October 2025.

[68] Bas Westerbaan and Luke Valenta. A look at the latest post-quantum signature standardization candidates. https://blog.cloudflare.com/another-look-at-pq-signatures/, November 2024.

[69] Xiang Xie, Kang Yang, Xiao Wang, and Yu Yu. Lightweight authentication of web data via garble-then-prove. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

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

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

[72] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. Zombie: Middleboxes that don't snoop. In *NSDI*. USENIX Association, 2024.

[73] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 270–282. ACM Press, October 2016.

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

A Cryptographic Preliminaries

A.1 Glossary

We encourage readers to reference the following three items when reading this work:

- Table 2 contains an list of symbols used in this work.
- Table 3 shows the cryptographic primitives used in TLS 1.3.
- Fig. 7 shows which TLS variables are held by the prover $\mathcal{P}$, verifier $\mathcal{V}$, or secret-shared in Ticket to Hide.

Symbol	Description
$\mathcal{V}$	Verifier
$\mathcal{P}$	Prover
$\mathcal{S}$	Server
N	Number of servers
sel	Index of selected server
$\leftarrow$	Variable assignment
$\leftarrow\!\!\$$	Uniform sampling
$[\![\ ]\!]$	Additive secret sharing
$\oplus$	Exclusive OR
Commit	Commitment
H	Hash function
g	Generator for a Diffie-Hellman group
q	Order of Diffie-Hellman group
GF	Galois field
CH	Client Hello message
ECH	Encrypted Client Hello message
SH	Server Hello message
EE	Encrypted Extensions message
Cert	Server Certificate message
CV	Certificate Verify message
SF	Server Finished message
CF	Client Finished message
req	Request message
resp	Response message
res_sec	Resumption secret
psk	Pre-shared key
ECHDE	Elliptic curve Diffie-Hellman ephemeral secret
hs_sec	Handshake secret
main_sec	Main secret
bkey	Binder key
CHTS / SHTS	Client/Server handshake traffic secret
CATS / SATS	Client/Server application traffic secret
k^C_{HS} / k^S_{HS}	Client/Server handshake encryption key
IV^C_{HS} / IV^S_{HS}	Client/Server handshake encryption IV
k^C_{App} / k^S_{App}	Client/Server application encryption key
IV^C_{App} / IV^S_{App}	Client/Server application encryption IV
$\mathcal{A}$	UC Adversary
$\mathcal{Z}$	UC Environment
Sim	UC Simulator

Table 2: Notation used in this paper.

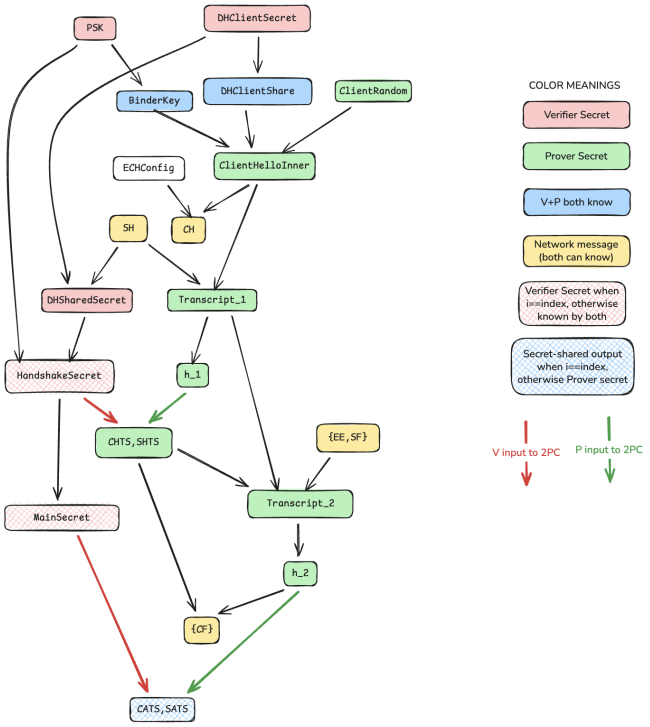


Figure 7: Diagram of variables in TLS 1.3, color-coded to show whether they are known by the verifier, prover, both, or neither in Ticket to Hide π_{TTH}. Black arrows denote which variables can be derived from other variables. Non-black arrows represent derivations that are performed in 2PC.

A.2 TLS 1.3

TLS is a protocol that allows web clients to connect to servers, authenticate their identity, and exchange end-to-end encrypted data, while preventing tampering and eavesdropping by malicious network adversaries. TLS 1.3 is the most recent version of TLS, and is substantially different from TLS 1.2. Below we describe the TLS 1.3 protocol. We depict the protocol in Fig. 8.

A.2.1 Cryptographic Assumptions. We make the standard assumptions about the cryptographic primitives used within TLS: we assume that the Advanced Encryption Standard (AES) is a pseudorandom permutation family, we model HMAC has a random oracle, and we assume that AES-GCM is an authenticated encryption scheme. We emphasize that AES-GCM is not a *committing* encryption scheme [19, 29]; for this reason, the Ticket to Hide protocol explicitly includes commitments to the application keys.

A.2.2 Basic TLS Handshake. The basic TLS handshake is depicted in steps 1-6 of Figure 8. A TLS handshake begins with a client C sending a server $\mathcal{S}$ a Client Hello (CH) message (step 1). This message contains many data fields, including:

- A list of *ciphersuites*: Symbols denoting cryptographic primitives that the client supports. For example, the ciphersuite `TLS_AES_128_GCM_SHA256` uses 128 bit AES-GCM encryption and SHA256 as a hash function.

Syntax	Description
Symmetric Primitives	
$\mathsf{Enc}(k, m) \rightarrow c$	Encrypts message m using key k to produce ciphertext c
$\mathsf{Dec}(k, c) \rightarrow m$	Decrypts c using key k
Commitments	
$\mathsf{Commit}(m) \rightarrow (\mathsf{com}, \mathsf{decom})$	Randomized algorithm to produce a new commitment to message m
$\mathsf{Open}(\mathsf{com}, \mathsf{decom}) \rightarrow m$ or $\perp$	Opens the given commitment to the original message m, or returns $\perp$ to indicate an invalid commitment/opening.
TLS Key Schedule	
CHSecrets(psk) $\rightarrow$ (csecret, cshare, bkey)	Chooses secrets needed to build a ClientHello, namely the client's key share and the binder key.
MainSecrets(psk, csecret, svrMsg1) $\rightarrow$ hssec, mainsec	Runs TLS key derivation to derive the handshake and main secretes form the PSK, client private key share, and server public key share.
TLS Messages	
ClRnd1(tikid, bkey, cshare, echconfig) $\rightarrow$ clMsg1	Constructs a ClientHello message using ECH and ticket resumption
CheckRnd1(hssec, clMsg1, svrMsg1) $\rightarrow$ state	Using the client and server handshake messages (from CH to SV), plus the derived handshake key, checks that the handshake decrypts properly, indicates PSK and key share acceptance, and has the correct SF.
ClRnd2(state, mainsec, req) $\rightarrow$ clMsg2	Computes the second flight from the client, which includes the CF message to end the handshake, plus an encrypted application data record encoding the req
ClDec(state, mainsec, svrMsg2) $\rightarrow$ resp	Decrypts the last record back from the server

Table 3: Cryptographic and TLS primitives used in this paper.

- A list of key exchange groups and elements. Ciphersuites do not specify a key exchange protocol, so the client lists which the key exchange groups they support separately. For each supported group, they include a key exchange element to use.
- Client randomness: 32 bytes, uniformly sampled.

The client may include many other additional extensions in the CH message (see [53]).

Upon receiving the CH message, the server responds with their their own Server Hello (SH) message (step 2). Like the CH, this message includes many data fields, such as:

- A single ciphersuite, selected from the client's list. The client and server will use this ciphersuite going forward.
- A single key exchange group and element. The group is selected from the client's list, and the element is sampled according to the key exchange mechanism.
- Server randomness: 32 bytes, uniformly sampled.

Upon generating a key exchange element, the server can now generate a shared secret via the key exchange protocol. They will use this, and the current transcript hash $h_1 = H(\mathsf{CH}, \mathsf{SH})$ to generate handshake traffic keys. These are symmetric authenticated encryption keys. The client and server use different keys, but both are computed similarly. The server will encrypt subsequent handshake messages under the server handshake encryption key ($\mathsf{k}^{\mathsf{S}}_{\mathsf{HS}}$). First, they may include some encrypted extensions to the SH message. We denote these as EE. There are many possible encrypted extensions, but we will not detail them here as none are relevant for our protocol.

The server now sends a TLS certificate. This is a blob of data containing the server's signature verification key, and a series of signatures. The idea of the certificate is to convince the client that the provided verification key is the legitimate key of the server. To do this, the server gets their public key signed by other entities, who in turn may get their public key signed by others. At the top level is a Certificate Authority (CA), a trusted signing party. The server must send an unbroken chain of signatures from their key to a signature by a CA. By verifying these signatures, the client can be convinced that the server's key was not tampered with.

The server sends a Certificate Verify (CV) message. This message contains a signature on the current transcript hash $H(\mathsf{CH}, \mathsf{SH}, \mathsf{EE}, \mathsf{Cert})$, signed by the server's private key. The transcript hash here is computed using the *plaintext* transcript.

Additionally, the server sends a Server Finished (SF) message. This consists of an HMAC of the current transcript hash $H(\mathsf{CH}, \ldots, \mathsf{CV})$, using a "finished key" fk as the key. The fk is derived similarly to handshake keys.

The client now receives the SH and encrypted EE, Cert, CV, SF messages. Upon receiving the SH, they can derive the shared secret and compute the same handshake keys as the server, allowing them to decrypt the subsequent messages. They verify the signatures in the certificate and CV message, and re-derive the SF value, comparing it against the value they received from the server. The client now computes and sends their own Client Finished (CF) message (step 3), which is computed similarly to the SF message, and encrypted under the client handshake key.

Now, the client uses the current transcript hash $H(\mathsf{CH}, \ldots, \mathsf{CF})$ to derive a new set of encryption keys. These are the application keys, which will be used to encrypt subsequent messages. The client

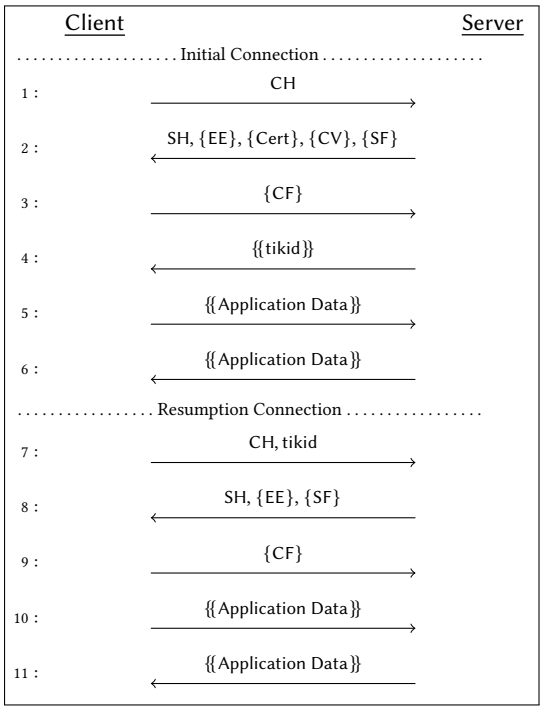


Figure 8: The TLS 1.3 handshake and ticket resumption protocols. Steps 1-6 follow the standard TLS 1.3 handshake procedure. Steps 7-11 depict a resumption handshake using a previously established ticket. Values in single braces {} are encrypted under handshake keys, and values in double braces {{}} are encrypted under application keys.

may immedately start sending encrypted application data under this key.

Upon receiving and verifying the CF message, the server can also derive the same application encryption keys. They generate a resumption ticket (described in the next section) and send it to the verifier, encrypted under the server application key (step 4). Additionally, both client and server compute a resumption secret (res_sec), derived in a similar manner as the handshake and application keys, using the current transcript hash. Now, the client and server may exchange encrypted application data (steps 5-6). This may continue for many rounds.

A.2.3 TLS 1.3 Handshake with Ticket Resumption. Subsequent connections between the same client and server can be done with less communication using the res_sec and ticket computed in the initial connection. This process is depicted in steps 7-11 of Figure 8.

Upon computing the res_sec in the initial connection, both client and server will derive the same *pre-shared key* value psk by combining res_sec and the ticket value in a HKDF. The client and server both store the ticket and psk values from in the initial connection past the lifespan of the initial TLS connection, for up to a few days. Alternatively, the server may avoid storing anything long-term by setting the ticket to be an encryption of the psk and other session identifiers under a server-owned key.

The client initiates a reconnection by sending a Client Hello message to the server (step 7) that contains a `pre_shared_key` extension—an unencrypted extension that includes the ticket and a *binder* value. The binder is an HMAC of the hash of the CH message, under a *binder key*—a value derived from the psk. The binder serves as a demonstration that the client knows the psk. The server has stored the psk (or alternatively recovers it by decrypting the ticket), and can re-compute the binder and compare it against the client's value.

Upon verifying the binder, the server responds with a Server Hello message (step 8). This SH message contains its own `pre_shared_key` extension, which includes a binder value computed by the server. This value is an HMAC of the current transcript hash $H(\mathsf{CH}, \mathsf{SH})$, encrypted under the same binder key. The client can similarly check the server's binder by recomputing the correct value. In doing so, they verify the server's identity. The binder can only be computed with the psk (except with negligible probability), and the psk was securely derived from the main_sec of the initial connection. During that connection, the client verified the server's certificate and Certificate Verify message, convincing them they were communicating with the legitimate server. The main_sec can only be derived by the client and that server (by the security of the key exchange protocol). Therefore, a valid binder value could *only* be produced by the previously authenticated server.

At this point, the client and server will compute a shared secret based on the key exchange values exchanged in the CH and SH of the resumption connection. The parties now compute handshake keys. In the reconnection, handshake and application keys are computed slightly differently than regular TLS 1.3: the psk is used as an additional input (see Figure 2). In a normal handshake, an all-zero string is used in place of the psk.

The server sends encrypted extensions and a Server Finished message, as in the regular TLS handshake. They do not send a Certificate or Certificate Verify message, as the client has already verified their identity via the binder value. From here, the handshake proceeds as in regular TLS 1.3. The client sends a Client Finished message, client and server generate application keys, and exchange encrypted application data (steps 9-11).

A.3 Definitions of Crypto Primitives

In this section, we provide definitions of common cryptographic primitives used in this work.

Commitments. A commitment scheme contains two algorithms:

- $\mathsf{Commit}(m) \rightarrow (\mathsf{com}, \mathsf{decom})$ is a randomized algorithm that receives a message m as input, and provides a commitment string com and decommitment randomness decom as outputs.
- $\mathsf{Open}(\mathsf{com}, \mathsf{decom}) \rightarrow m$ or $\perp$ receives the purported com and decom strings, and it verifies whether they are consistent with the message m.

A commitment scheme allows a committer to reveal a message a later time, and it provides two properties (binding and hiding) in between the message commitment and opening. The definition below is adapted from Katz-Lindell [34, Def. 6.13], and we consider both properties against computationally-bound adversaries.

- *Message binding*: It is computationally infeasible for a malicious committer to find a commitment string com along with two message/decommitment pairs (m', decom') and (m'', decom''), where $|m'| = |m''|$ but $m' \neq m''$, such that the commitment com can be opened to both messages:

$$\text{Open}(\text{com}, \text{decom}') \to m'$$
$$\text{Open}(\text{com}, \text{decom}'') \to m''.$$

- *Message hiding*: It is computationally infeasible for a malicious receiver to determine the message prior to opening, even in a chosen message attack. Formally, this game proceeds in two steps. First, the adversary chooses two strings $(m_0, m_1) \leftarrow \mathcal{A}_1$. Second, a bit is chosen uniformly at random $b \leftarrow\!\!\$\ \{0,1\}$, a commitment $(\text{com}, \text{decom}) \leftarrow \text{Commit}(m_b)$ is calculated, and the adversary attempts to guess b from the commitment string: $b' \leftarrow \mathcal{A}_2(m_0, m_1, \text{com})$. Then, the adversary has only negligible advantage at guessing b: that is, $\Pr[b = b'] \leq 1/2 + \text{negl}$.

Secure two-party computation. A secure two-party computation (2PC) scheme allows parties P_1 and P_2, each possessing a secret input, to evaluate a function f on their joint inputs, while revealing nothing to each other beyond the outputs of the function.

Formally, 2PC security is defined in a real/ideal world paradigm. In the real world, the parties execute an interactive cryptographic protocol π. In the ideal world, the parties give their secret inputs to a trusted third party, who computes the function and gives the correct outputs to both parties. We say that the protocol π is secure against static semi-honest adversaries if there exists an efficient *simulator* program Sim that, upon being given party i's input and correct output, can produce a "view" of party i in an execution of π (consisting of all messages and any internal randomness) that is computationally indistinguishable from a view of a real execution.

We refer the reader to Lindell's survey of simulation-based security [42] for more details, along with a formal definition for 2PC with security against malicious adversaries.

Zero knowledge proofs. A zero-knowledge (ZK) proof is a protocol that allows a prover to convince a verifier that some statement is true, without revealing any additional information [25]. Formally, a ZK proof protocol for an NP relation $R : \mathcal{X} \times \mathcal{W} \to \{0,1\}$ is an interactive protocol between a prover $\mathcal{P}$ and verifier $\mathcal{V}$ satisfying the following properties:

- *Completeness*: For all $x \in \mathcal{X}, w \in \mathcal{W}$ such that $R(x,w) = 1$, when an honest $\mathcal{P}$ and $\mathcal{V}$ perform the protocol on public input x, $\mathcal{V}$ outputs 1 (except with at most negligible probability).
- *Soundness*: For any $x \in \mathcal{X}$ such that for all $w \in \mathcal{W}, R(x,w) = 0$, when a malicious $\mathcal{P}$ and $\mathcal{V}$ perform the protocol on public input x, the probability that $\mathcal{V}$ outputs 1 is negligible.
- *Zero-Knowledge*: There exists an efficient simulator Sim such that for any efficient, malicious verifier $\mathcal{V}^*$, the simulator can generate a view computationally indistinguishable from that of $\mathcal{V}^*$ interacting with an honest prover, for any public input $x \in L_R$.

An alternative definition in the UC framework is given in Fig. 11.

B Additional Attack on DiStefano

In this section we describe an attack on the original DiStefano protocol, as described in the conference version of the paper [13]. We disclosed this attack to the authors in January 2026. Subsequently, they published an updated version of their paper that is secure against this attack [12]. We describe our attack and their mitigation here, for historical record.

Our attack allows a malicious prover to produce commitments to arbitrary data and convince a verifier they originated from a given server. Celi et al. additionally describe a variant of the DiStefano protocol which does not require a zero-knowledge proof of signature verification, at the cost of revealing the identity of the prover's chosen server to the verifier (akin to the single-server setting). This version of the protocol is not vulnerable to our attack.

First, we summarize the three-party handshake of the original DiStefano protocol. The prover and verifier each begin by generating an elliptic curve group element g^x and g^z, respectively. The verifier sends their share to the prover, who uses g^{x+z} as the client key share in a Client Hello message to the real server. DiStefano does not hide network traffic, so the prover only communicates with the real server and does not make any decoy connections. The server responds with a Server Hello message containing a group element g^y, which the prover sends to the verifier. The prover and verifier then compute g^{xy} and g^{zy} respectively, and use an interactive ECtF protocol to compute additive shares of the x-coordinate of the key exchange secret, $g^{(x+z)y}$. From this, they use authenticated garbling [66] to compute secret shares of the SHTS, CHTS, and handshake keys, as well as the finished key fk (used to compute the Server Finished message), which is revealed to both $\mathcal{P}$ and $\mathcal{V}$. The prover then forwards the TLS transcript to the verifier, which at this point contains messages up to the Server Finished message. The verifier reveals their SHTS share to the prover, who can now decrypt the server's encrypted handshake messages. The prover now computes transcript hashes $H_2 = H(\mathsf{CH} \ldots \mathsf{SF})$, $H_3 = H(\mathsf{CH} \ldots \mathsf{Cert})$ and $H_4 = H(\mathsf{CH} \ldots \mathsf{CV})$, and verifies that the server's CV message encrypts a valid signature on H_3. The prover then generates a ZK proof π, of knowledge proving that they know a signature on H_3 that verifies under one of the N server public keys (agreed upon *a priori*). They send π, H_2, H_3, and H_4 to $\mathcal{V}$. The verifier now checks the Server Finished message by verifying that $\mathsf{SF} = \mathsf{HMAC}(\mathsf{fk}, H_4)$, and validates the prover's ZK proof, using H_3 and the set of server public keys as public inputs. They abort if either of these checks fail. Prover and verifier now use H_2 to jointly compute secret shares of the TLS application keys under MPC. This concludes the three-party handshake. From here, the prover and verifier jointly compute an encrypted client request in a similar fashion to our protocol.

The issue with this protocol is that the verifier cannot validate that H_2, H_3, and H_4 are correct hashes of the right TLS transcript. These hashes are (meant to be) computed over the *decrypted* TLS transcript, which the verifier never learns. Therefore, they have no way to detect if the prover sends an incorrect hash value.

This opens the door for the following attack. A malicious prover can first make a regular TLS 1.3 connection with any server in the approved set, and record the server signature and the H_3 hashes computed during the handshake. During the three-party handshake, the prover will impersonate all server messages. They construct

a Server Hello message by sampling a discrete log y themselves and using g^y as the server's Diffie-Hellman group element. This allows $\mathcal{P}$ to compute the key exchange secret locally, which they can use to derive handshake keys and produce encrypted EE, Cert, CV, and SF messages on their own. In the CV message, they include the server's signature on H_3 from the previous connection. The prover can now send this encrypted transcript to the verifier, and will later send the H_3 hash from the previous connection to the prover alongside a correctly constructed ZK proof π that they know a signature from one of the servers on H_3. They compute H_4 as the hash of the impersonated transcript. They can choose any value for H_2, since it is not validated. The prover sends π, H_2, H_3, and H_4 to the verifier. The verifier will now check that $\mathsf{SF} = \mathsf{HMAC}(\mathsf{fk}, H_4)$, which succeeds since SF and H_4 were computed in the same way an honest server would compute them. Further, the proof π verifies successfully, since the witness is indeed a valid signature from one of the servers on H_3. From here, the verifier will continue the protocol as normal, convinced that the prover is communicating with an approved server. However, the prover knows all TLS secrets, and can compute the application keys on their own. This allows them to encrypt arbitrary data under the server application keys, and commit to the ciphertexts as if they came from the real server. Since these ciphertexts are encrypted under the "real" application keys, the prover can successfully decrypt them in ZK during the decommitment/proof phase.

An updated version of the DiStefano paper describes a mitigation to this attack [12]. Their solution is to extend the zero-knowledge proof π of signature verification to additionally prove that H_2, H_3, and H_4 are hashes of the correct transcript, thereby preventing the prover from replaying a hash from a previous connection. The drawback is that the ZK proof has increased complexity, as it requires computing the hash of a several kilobyte-long string inside the proof.

C 2PC AES-GCM Encryption

The GCM authentication tag [48] is defined as follows: Let C be the ciphertext, expressed in n 128-bit chunks $C_1, \ldots C_n$. Let A be the authentication data, expressed in m 128-bit chunks $A_1, \ldots, A_m$. Let $S \in \mathsf{GF}(2^{128})^{m+n+2}$ be the concatenation $A \parallel C \parallel m \parallel n$, padded appropriately. For a key k and initial value IV, let $Z = \mathsf{Enc}(k, \mathsf{IV})$ and $H = \mathsf{Enc}(k, 0^{128})$. The GCM tag is defined as

$$\mathsf{GCMTag}(k, C, A, \mathsf{IV}) = Z \oplus \sum_{i=1}^{m+n+2} S_i \cdot H^{m+n+3-i},$$

where addition and multiplication operations are done in the field $\mathsf{GF}(2^{128})$.

To compute this function under 2PC, we do the following. First, $\mathcal{P}$ and $\mathcal{V}$ use a garbled circuit to compute the unauthenticated ciphertext C on $\mathcal{P}$'s secret plaintext, revealing the output to $\mathcal{P}$. Second, $\mathcal{P}$ and $\mathcal{V}$ compute Z and H in a garbled circuit, each learning a share of the results. Then, $\mathcal{P}$ and $\mathcal{V}$ use Oblivious Linear Execution (OLE) to convert their share of H to multiplicative shares, allowing them to locally compute sharings of all powers of H. $\mathcal{P}$ additionally multiplies their share of each power of H powers with a block of S, thereby producing a sharing of $S_i \cdot H^{m+n+3-i}$ for each $i \in [m+n+2]$. The parties use OLE again to convert their multiplicative shares back to additive shares, and each locally adds their shares together to complete the polynomial evaluation. The parties additionally add their share of Z to this, resulting in additive shares of $\mathsf{GCMTag}(k, C, A, \mathsf{IV})$.

This differs slightly from the approach of Xie et al. [69]. In their protocol, S is revealed to both parties. $\mathcal{P}$ does not multiply their multiplicative sharings of the powers of H with blocks of S; instead, $\mathcal{P}$ and $\mathcal{V}$ use OLE to obtain additive shares of the H powers, and then scalar multiply these shares locally with the blocks of S.

$\mathcal{F}_{\mathsf{TCP}}$ has no sid and responds to the following messages:

- ($\textsc{Connect}$, $\mathsf{pid}_{\mathcal{S}}$) from pid_C:
  (1) Leak ($\textsc{Connecting}$, pid_C, $\mathsf{pid}_{\mathcal{S}}$) to $\mathcal{A}$ and wait for response (sessionId)
  (2) Return ($\textsc{AckConnect}$, sessionId) to pid_C
- ($\textsc{Transmit}$, sessionId, payload) from pid_x:
  (1) Leak ($\textsc{Sending}$, sessionid, payload, pid_x) to $\mathcal{A}$ and wait for response (OK)
  (2) Return ($\textsc{AckSend}$) to pid_x
- Backdoor ($\textsc{Deliver}$, sessionId, payload, pid_x) from $\mathcal{A}$:
  (1) Send ($\textsc{TCPMessage}$, sessionId, payload) to pid_x

Figure 9: Ideal functionality $\mathcal{F}_{\mathsf{TCP}}$

$\mathcal{F}_{\mathsf{TLS}}$ has sid = ($\mathsf{pid}_{\mathcal{S}}$, psklen) and responds to the following messages:

- ($\textsc{GetTicket}$, sid) from pid_C:
  (1) Generate a random byte string psk with size psklen.
  (2) Send ($\textsc{TicketRequest}$, sid, psk, pid_C) to $\mathsf{pid}_{\mathcal{S}}$ and wait for response (tikid)
  (3) Leak ($\textsc{TicketSent}$, sid, pid_C) to $\mathcal{A}$
  (4) Return (tikid, psk) to pid_C
- ($\textsc{GetECH}$, sid) from pid_C:
  (1) Send ($\textsc{ECHRequest}$, sid, pid_C) to $\mathsf{pid}_{\mathcal{S}}$ and wait for response (echconfig).
  (2) Leak ($\textsc{ECHSent}$, sid, pid_C) to $\mathcal{A}$
  (3) Return (echconfig) to pid_C
- ($\textsc{RecordReqResp}$, sid, req, resp) from $\mathsf{pid}_{\mathcal{S}}$:
  (1) Append (sid, req, resp) to the saved list
  (2) Return (OK) to $\mathsf{pid}_{\mathcal{S}}$

Figure 10: Ideal functionality $\mathcal{F}_{\mathsf{TLS}}$

$\mathcal{F}_{\mathsf{ZK}}$ is parameterized by a relation R, has sid = ($\mathsf{pid}_{\mathcal{P}}$, $\mathsf{pid}_{\mathcal{V}}$nonce), and responds to the following message:

- ($\textsc{ZKProve}$, sid, x, w) from $\mathsf{pid}_{\mathcal{P}}$:
  (1) If $R(x, w) = \top$, then send ($\textsc{ZKProof}$, sid, x) to $\mathsf{pid}_{\mathcal{V}}$ and Halt
  (2) Otherwise Halt

Figure 11: Ideal functionality $\mathcal{F}_{\mathsf{ZK}}$, copied from Fig. 6 in [11].

$\mathcal{F}_{\text{2PC}}$ has sid = $(\text{pid}_{\mathcal{P}}, \text{pid}_{\mathcal{V}}, C^{(1)}, C^{(2)}, \text{nonce})$ and responds to the following messages:

- (Garble, sid, $\mathbf{w}^{(1)}$, cheats) from $\text{pid}_{\mathcal{P}}$:
  (1) If (initialDone, sid, ...) is recorded, Abort.
  (2) If $\text{pid}_{\mathcal{P}}$ is corrupt, parse cheats as $(C^{(\text{fail})}, C^{(\text{leak})})$
  (3) Else set $C^{(\text{fail})}$ and $C^{(\text{leak})}$ to always return $\perp$
  (4) Send (EvalInitial, sid) to $\text{pid}_{\mathcal{V}}$ and wait for response $(\mathbf{x})$
  (5) If $C^{(\text{fail})}(\mathbf{x}) = \top$, Abort
  (6) Else If $C^{(\text{leak})}(\mathbf{x}) = \top$, then
    (a) Record (initialDone, sid, leaked)
    (b) Return (Leaked, $\mathbf{x}$) to $\text{pid}_{\mathcal{P}}$
  (7) Else
    (a) Compute $\mathbf{y}^{(1)} \leftarrow C^{(1)}(\mathbf{w}^{(1)}, \mathbf{x})$
    (b) Record (initialDone, sid, $\mathbf{w}^{(1)}, \mathbf{x}$)
    (c) Return $(\mathbf{y}^{(1)})$ to $\text{pid}_{\mathcal{P}}$
- (FinishEval, sid, $\mathbf{w}^{(2)}$) from $\text{pid}_{\mathcal{P}}$:
  (1) If (evalDone, sid, ...) is recorded, Abort.
  (2) Lookup (initialDone, sid, $\mathbf{w}^{(1)}, \mathbf{x}$); Abort if not found
  (3) Send (EvalFinish, sid) to $\text{pid}_{\mathcal{V}}$ and wait for response (OK)
  (4) If $\mathbf{w}^{(1)}$ = leaked, then
    (a) Record (evalDone, sid, leaked)
    (b) Return () to $\text{pid}_{\mathcal{P}}$
  (5) Else
    (a) Compute $\mathbf{y}^{(2)}, \mathbf{z} \leftarrow C^{(2)}(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{y})$
    (b) Record (evalDone, sid, $\mathbf{x}, \mathbf{z}$)
    (c) Return $(\mathbf{y}^{(2)})$ to $\text{pid}_{\mathcal{P}}$
- (Release, sid, $\mathbf{x}'$) from $\text{pid}_{\mathcal{P}}$:
  (1) If (releaseDone, sid) is recorded, Abort.
  (2) Lookup (evalDone, sid, $\mathbf{x}, \mathbf{z}$); Abort if not found
  (3) If $\mathbf{x}$ = leaked or $\mathbf{x} \neq \mathbf{x}'$, Abort.
  (4) Record (releaseDone, sid)
  (5) Send (Verified, sid, $\mathbf{z}$) to $\text{pid}_{\mathcal{V}}$

Figure 12: Ideal functionality $\mathcal{F}_{\text{2PC}}$

D Ideal functionalities

We give here the full listing of the ideal functionalities referenced in §4. Concretely: Figs. 9 to 13 depict $\mathcal{F}_{\text{TCP}}$, $\mathcal{F}_{\text{TLS}}$, $\mathcal{F}_{\text{ZK}}$, $\mathcal{F}_{\text{2PC}}$, and $\mathcal{F}_{\text{MSO}}$, respectively.

E Formal Protocols for Ticket to Hide Parties

In §5 we gave a high-level description of our main Ticket to Hide protocol. Here we formally define the protocol π_{TTH} in the UC setting, in the hybrid world with local functionalities $\mathcal{F}_{\text{2PC}}$ and $\mathcal{F}_{\text{ZK}}$ and global functionalities $\mathcal{F}_{\text{TCP}}$ and $\mathcal{F}_{\text{TLS}}$.

Formally, the protocol is a monolith that controls all honest parties' interactions in the hybrid world. But informally, we can think of it as being separated into the three roles that parties take in our framework: Prover, Verifier, or Server. Accordingly, the protocol is described in Figs. 14 to 16. Note that the prover and verifier protocols are consistent with Fig. 1, and the bulk of the server protocol (Fig. 16) is running the TLS 1.3 protocol directly.

F Detailed Security Analysis for Ticket to Hide

In this section, we prove Theorem 5.1. For convenience, we reproduce the theorem statement below.

Theorem 5.1. *Based on the cryptographic assumptions listed in Appendix A.2.1 (which are commonly used in TLS itself), The Ticket to Hide protocol π_{TTH} (Fig. 1 and Figs. 14 to 16) along with subroutines $\mathcal{F}_{\text{2PC}}$ and $\mathcal{F}_{\text{ZK}}$ UC-realizes functionality $\mathcal{F}_{\text{MSO}}$ (Fig. 13) in the presence of global functionalities $\mathcal{F}_{\text{TCP}}$ and $\mathcal{F}_{\text{TLS}}$ with static corruptions.*

For the security proof, we need to provide a simulator Sim that acts as a translator in the ideal world between corrupted parties and the ideal functionality. In the real world, there is no simulator; the real adversary $\mathcal{A}$ sends actual network messages and interacts with sub-functionalities $\mathcal{F}_{\text{2PC}}$ and $\mathcal{F}_{\text{ZK}}$ just like our actual protocol does. The goal of Sim is to simulate the adversary's real-world interactions with honest parties, using only information gained from the ideal functionality and from the adversary itself. The proof succeeds if the Environment $\mathcal{Z}$ — who controls $\mathcal{A}$ and also all input to and output from honest users — cannot distinguish between the real/hybrid and ideal worlds.

The full listing of our simulator for π_{TTH} is given in Fig. 17. We now explain why this provides a proof of the theorem.

There are essentially two cases to consider for the proof: when the Verifier is corrupted, and when the Prover is corrupted. (If both are corrupted, there is nothing to do, and if neither is corrupted, there is no extra leakage to simulate.)

F.1 Corrupted Verifier

When $\mathcal{V}$ is corrupted but $\mathcal{P}$ is honest, the simulator Sim must simulate the real $\mathcal{P}$'s interactions with $\mathcal{V}$ in the actual protocol π_{TTH}, using only information made available to the corrupted verifier by the ideal functionality $\mathcal{F}_{\text{MSO}}$.

Crucially, Sim cannot interact with the servers directly, because those are global functionalities and any extra messages sent to them in the ideal world would break indistinguishability. However, as $\mathcal{F}_{\text{2PC}}$ is a local functionality, Sim directly reads the (corrupted) verifier's inputs to the 2PC, which is needed for the simulator to succeed.

When Sim receives a network message with $(\text{tikid}_i, k_i^B, g^{\alpha_i})_{i \in [N]}$, from corrupted $\text{pid}_{\mathcal{V}}$, it invokes the online phase of the ideal functionality $\mathcal{F}_{\text{MSO}}$ to begin the protocol in the ideal world.

Recall that, when $\mathcal{V}$ is corrupted, the ideal functionality will use the provided binder keys and key shares to produce the initial ECH messages to all servers. The simulator Sim can easily provide this simply by passing along those values directly from the adversary. This ensures that the first-round messages to/from the servers in the ideal world are identical to what they would be in the real world with an honest prover.

There are two steps when Sim will be required to produce more information by the ideal protocol: after the handshakes (first round), and after the requests/replies (second round).

After the handshake messages. After the first round of messages with the servers, $\mathcal{F}_{\text{MSO}}$ sends all of the SH messages to Sim. (Notice this essentially leaks nothing new since the plaintext SH messages were already leaked when they were transmitted via $\mathcal{F}_{\text{TCP}}$.)

Now Sim interacts with $\mathcal{A}$ as if continuing the real protocol: Sim

$\mathcal{F}_{\text{MSO}}$ is parameterized by a commitment scheme (Commit, Open), has sid = $(\text{pid}_{\mathcal{P}}, \text{pid}_{\mathcal{V}}, \text{sid}_1^{\text{TLS}}, \ldots, \text{sid}_N^{\text{TLS}}, \text{nonce})$, and responds to the following messages:

- (ProverPrep, sid) from $\text{pid}_{\mathcal{P}}$:
  (1) If (proverPrepped, sid, ...) already recorded, Abort
  (2) For each $i \in [n]$, send (GetECH, $\text{sid}_i^{\text{TLS}}$) to $\mathcal{F}_{\text{TLS}}$ and wait for response (echconfig_i)
  (3) Record (proverPrepped, sid, $\text{echconfig}_1, \ldots, \text{echconfig}_n$)
  (4) Return (OK) to $\text{pid}_{\mathcal{P}}$
- (VerifierPrep, sid) from $\text{pid}_{\mathcal{V}}$:
  (1) If (verifierPrepped, sid, ...) already recorded, Abort
  (2) For each $i \in [N]$, send (GetTicket, $\text{sid}_i^{\text{TLS}}$) to $\mathcal{F}_{\text{TLS}}$ and wait for response $(\text{tikid}_i, \text{psk}_i)$
  (3) Record (verifierPrepped, sid, $(\text{tikid}_i, \text{psk}_i)_{i\in[N]}$)
  (4) Return (OK) to $\text{pid}_{\mathcal{V}}$
- (OnlinePhase, sid, optionalAdvInfo) from $\text{pid}_{\mathcal{V}}$:

---------- Handle prover corruption ----------

(1) If $\text{pid}_{\mathcal{P}}$ is corrupt, then
  (a) Leak (GetReqResp, sid, $(\text{tikid}_i, \text{psk}_i)_{i\in[N]}$) to $\mathcal{A}$ and wait for response (com, decom)
  (b) Set (req, resp) $\leftarrow$ Open(com, decom)
  (c) If (req, resp) $\neq \perp$, and there exists $i \in [N]$ such that $(\text{sid}_i^{\text{TLS}}, \text{req}, \text{resp})$ is in the memory of $\mathcal{F}_{\text{TLS}}$, then return (com) to $\text{pid}_{\mathcal{V}}$
  (d) Else Abort

---------- Prepare inputs ----------

(2) If $\text{pid}_{\mathcal{V}}$ is corrupt, then parse optionalAdvInfo as $(\text{cshare}_i, \text{bkey}_i, \text{tikid}_i)_{i\in[N]}$
(3) Else if $\text{pid}_{\mathcal{V}}$ is not corrupt, do
  (a) Lookup (verifierPrepped, $(\text{tikid}_i, \text{psk}_i)_{i\in[N]}$); Abort if not found.
  (b) For $i \in [N]$, set $\text{csecret}_i, \text{cshare}_i, \text{bkey}_i \leftarrow \text{CHSecrets}(\text{psk}_i)$
(4) Lookup (proverPrepped, $(\text{echconfig}_i)_{i\in[N]}$); Abort if not found.
(5) Send (GetRequests, sid) to $\text{pid}_{\mathcal{P}}$ and wait for response $(\text{sel}, \text{req}_1, \ldots, \text{req}_N)$

---------- First round (handshake) ----------

(6) For $i \in [N]$, do
  (a) $\text{clMsg1}_i \leftarrow \text{ClRnd1}(\text{tikid}_i, \text{bkey}_i, \text{cshare}_i, \text{echconfig}_i)$
  (b) Send (Connect, $\text{pid}_{\mathcal{S}_i}$) to $\mathcal{F}_{\text{TCP}}$ and wait for response (sessionId_i)
  (c) Send (Transmit, sessionId, clMsg1_i) to $\mathcal{F}_{\text{TCP}}$
  (d) Wait for message (TCPMessage, $\text{sessionId}_i, \text{svrMsg1}_i$) from $\mathcal{F}_{\text{TCP}}$

---------- Detect first round cheating ----------

(7) If $\text{pid}_{\mathcal{V}}$ is corrupt, then leak (Round1Done, sid, $(\text{svrMsg1}_i)_{i\in[N]}$) to $\mathcal{A}$ and wait for response $((\text{hssec}_i, \text{mainsec}_i)_{i\in[N]})$
(8) Else for $i \in [N]$, set $\text{hssec}_i, \text{mainsec}_i \leftarrow \text{MainSecrets}(\text{psk}_i, \text{csecret}_i, \text{svrMsg1}_i)$
(9) For $i \in [N]$, set $\text{state}_i \leftarrow \text{CheckRnd1}(\text{hssec}_i, \text{clMsg1}_i, \text{svrMsg1}_i)$
(10) If any $\text{state}_i = \perp$, then send (FailRound1) to $\text{pid}_{\mathcal{P}}$ and Abort.

---------- Second round (req/resp) ----------

(11) For $i \in [N]$, do
  (a) $\text{clMsg2}_i \leftarrow \text{ClRnd2}(\text{state}_i, \text{mainsec}_i, \text{req}_i)$
  (b) Send (Transmit, sessionId, clMsg2_i) to $\mathcal{F}_{\text{TCP}}$
  (c) Wait for message (TCPMessage, $\text{sessionId}_i, \text{svrMsg2}_i$) from $\mathcal{F}_{\text{TCP}}$
  (d) $\text{resp}_i \leftarrow \text{ClDec}(\text{state}_i, \text{mainsec}_i, \text{svrMsg2}_i)$

---------- Final commitments ----------

(12) com, decom $\leftarrow$ Commit$((\text{req}_{\text{sel}} \| \text{resp}_{\text{sel}}))$
(13) If $\text{pid}_{\mathcal{V}}$ is corrupt, then
  (a) Leak (Round2Done, sid, com) to $\mathcal{A}$ and wait for reply (shouldFail)
  (b) If shouldFail is $\top$, send (FailRound2) to $\text{pid}_{\mathcal{P}}$ and Abort.
(14) Send (Replies, sid, decom, $\text{resp}_1, \ldots, \text{resp}_N$) to $\text{pid}_{\mathcal{P}}$ and wait for response (OK)
(15) Return (com) to $\text{pid}_{\mathcal{V}}$

Figure 13: Ideal functionality $\mathcal{F}_{\text{MSO}}$

grabs all server key shares g^{β_i} and sends them to $\mathcal{A}$, and then they begin the 2PC. Since $\mathcal{F}_{\text{2PC}}$ is a local subroutine, the simulator acts

Protocol π_{TTH}, in the context of a Prover, responds to the following inputs and messages:

- Input (ProverPrep, sid) from $\mathcal{Z}$:
  (1) If (eches, sid, ...) already recorded, Abort.
  (2) For $i \in [N]$ send (GetECH, $\mathsf{sid}_i^{\mathsf{TLS}}$) to $\mathcal{F}_{\mathsf{TLS}}$ and wait for response ($\mathsf{echconfig}_i$)
  (3) Record (eches, sid, $\mathsf{echconfig}_1, \ldots, \mathsf{echconfig}_N$) in local memory
  (4) Output (OK) to $\mathcal{Z}$
- Direct message (StartHS, sid, $(\mathsf{cshare}_i, \mathsf{bkey}_i, \mathsf{tikid}_i)_{i\in[N]}$) from $\mathsf{pid}_{\mathcal{V}}$:

  ---------- Prepare inputs ----------

  (1) Lookup (eches, $(\mathsf{echconfig}_i)_{i\in[N]}$) in local memory; Abort if not found.
  (2) Output (GetRequests, sid) to $\mathcal{Z}$ and wait for response input ($\mathsf{sel}, \mathsf{req}_1, \ldots, \mathsf{req}_N$)

  ---------- First round (handshake) ----------

  (3) For $i \in [N]$, do
    (a) $\mathsf{InnerCH}_i, \mathsf{clMsg1}_i \leftarrow \mathsf{ClRnd1}(\mathsf{tikid}_i, \mathsf{bkey}_i, \mathsf{cshare}_i, \mathsf{echconfig}_i)$
    (b) Send (Connect, $\mathsf{pid}_{\mathcal{S}_i}$) to $\mathcal{F}_{\mathsf{TCP}}$ and wait for response ($\mathsf{sessionId}_i$)
    (c) Send (Transmit, sessionId, $\mathsf{clMsg1}_i$) to $\mathcal{F}_{\mathsf{TCP}}$
    (d) Wait for message (TCPMessage, $\mathsf{sessionId}_i$, $\mathsf{svrMsg1}_i$) from $\mathcal{F}_{\mathsf{TCP}}$

  ---------- Run 2PC ----------

  (4) For $i \in [n]$, do
    (a) Parse $\mathsf{svrMsg1}_i$ as $\mathsf{SH}_i \| \mathsf{encEEandSF}_i$
    (b) Extract server key share sshare_i from $\mathsf{svrMsg1}_i$
    (c) Compute $h_{1,i} \leftarrow H(\mathsf{innerCH}_i \| \mathsf{SH}_i)$
  (5) Send direct message (SKeyShares, $\mathsf{sid}(\mathsf{sshare}_i)_{i\in[N]}$) to $\mathsf{pid}_{\mathcal{V}}$ and wait for response (OK)
  (6) Send (Garble, $\mathsf{sid}^{\mathsf{2PC}}$, $(\mathsf{sel}, h_{1,\mathsf{sel}}, \mathsf{req}_{\mathsf{sel}})$, ()) to $\mathcal{F}_{\mathsf{2PC}}$ and wait for response ($\mathsf{CHTS}_{\mathsf{sel}}, \mathsf{SHTS}_{\mathsf{sel}}$)
  (7) Compute $\mathsf{EE}_{\mathsf{sel}}, \mathsf{SF}_{\mathsf{sel}}, \mathsf{EncCF}_{\mathsf{sel}} \leftarrow \mathsf{ProcessHS}(\mathsf{SHTS}_{\mathsf{sel}}, \mathsf{encEEandSF}_{\mathsf{sel}})$
  (8) If ProcessHS returned $\perp$, set $\mathsf{hsFail} \leftarrow \top$ and $h_2 \leftarrow H(\$)$
  (9) Else compute $h_{2,\mathsf{sel}} \leftarrow H(\mathsf{innerCH}_{\mathsf{sel}} \| \mathsf{SH}_{\mathsf{sel}} \| \mathsf{EE}_{\mathsf{sel}} \| \mathsf{SF}_{\mathsf{sel}})$
  (10) Send (FinishEval, $\mathsf{sid}^{\mathsf{2PC}}$, $h_{2,\mathsf{sel}}$) to $\mathcal{F}_{\mathsf{2PC}}$ and wait for response $((\mathsf{hs_sec}_i, \mathsf{main_sec}_i)_{i\neq\mathsf{sel}}, [\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{P}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{P}}, \{\{\mathsf{req}_{\mathsf{sel}}\}\})$

  ---------- Detect first round cheating ----------

  (11) For $i \neq \mathsf{sel}$, set $\mathsf{state}_i \leftarrow \mathsf{CheckRnd1}(\mathsf{hssec}_i, \mathsf{clMsg1}_i, \mathsf{svrMsg1}_i)$
  (12) If $\mathsf{hsFail} = \top$ or any $\mathsf{state}_i = \perp$, then send (FailRound1) to $\mathsf{pid}_{\mathcal{P}}$ and Abort.

  ---------- Second round (req/resp) ----------

  (13) Set $\mathsf{clMsg2}_{\mathsf{sel}} \leftarrow \mathsf{EncCF} \| \{\{\mathsf{req}_{\mathsf{sel}}\}\}$
  (14) For $i \neq \mathsf{sel}$, set $\mathsf{clMsg2}_i \leftarrow \mathsf{ClRnd2}(\mathsf{state}_i, \mathsf{mainsec}_i, \mathsf{req}_i)$
  (15) For $i \in [N]$, do
    (a) Send (Transmit, sessionId, $\mathsf{clMsg2}_i$) to $\mathcal{F}_{\mathsf{TCP}}$
    (b) Wait for message (TCPMessage, $\mathsf{sessionId}_i$, $\mathsf{svrMsg2}_i$) from $\mathcal{F}_{\mathsf{TCP}}$
  (16) For $i \neq \mathsf{sel}$, set $\mathsf{resp}_i \leftarrow \mathsf{ClDec}(\mathsf{state}_i, \mathsf{mainsec}_i, \mathsf{svrMsg2}_i)$

  ---------- Final commitments and checks ----------

  (17) $\mathsf{com}, \mathsf{decom} \leftarrow \mathsf{Commit}([\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{P}} \| [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{P}} \| \{\{\mathsf{req}_{\mathsf{sel}}\}\}, \{\{\mathsf{resp}_{\mathsf{sel}}\}\} = \mathsf{svrMsg2}_{\mathsf{sel}})$
  (18) Send direct message (Commitment, sid, com) to $\mathsf{pid}_{\mathcal{V}}$ and wait for response $((\mathsf{csecret}_i, \mathsf{psk}_i)_{i\in[N]})$
  (19) Recompute all values using $\mathsf{csecret}_i$ and psk_i values and compare with previous computations. If any do not match, output (FailRound2) to $\mathcal{Z}$ and Abort.
  (20) Output (Replies, sid, decom, $\mathsf{resp}_1, \ldots, \mathsf{resp}_N$) to $\mathcal{Z}$ and wait for response input (OK)
  (21) Send (Release, $\mathsf{sid}^{\mathsf{2PC}}$, $(\mathsf{hs_sec}_i, \mathsf{main_sec}_i)^{i\in[N]}$) to $\mathcal{F}_{\mathsf{2PC}}$
  (22) Send (ZKProve, $\mathsf{sid}^{\mathsf{ZK}}$, $(\mathsf{com}, [\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{V}}), \mathsf{decom}$) to $\mathcal{F}_{\mathsf{ZK}}$

Figure 14: Prover $\mathcal{P}$'s part of Protocol π_{TTH}

as the $\mathcal{F}_{\mathsf{2PC}}$ box from the adversary's perspective; therefore Sim does not need to input anything, and then it directly receives all of the hssec_i and $\mathsf{mainsec}_i$ values which are input by $\mathcal{A}$.

Then Sim simply hands over these handshake secrets and main secrets, back to the ideal functionality $\mathcal{F}_{\mathsf{MSO}}$, to check the handshakes succeeded and compute the next round of messages.

Notice that $\mathcal{A}$ could of course input invalid handshake or main secrets, but if they do, it will cause the relevant part of the *ideal* execution in $\mathcal{F}_{\mathsf{MSO}}$ to fail. So Sim just needs to extract them from $\mathcal{F}_{\mathsf{2PC}}$ and pass them along to $\mathcal{F}_{\mathsf{MSO}}$ who is actually communicating with the servers.

Protocol π_{TTH}, in the context of a Verifier, responds to the following inputs and messages:

- Input (VerifierPrep, sid) from $\mathcal{Z}$:
  (1) If (tickets, sid, ...) already recorded, Abort
  (2) For each $i \in [N]$, send (GetTicket, $\mathsf{sid}_i^{\mathsf{TLS}}$) to $\mathcal{F}_{\mathsf{TLS}}$ and wait for response $(\mathsf{tikid}_i, \mathsf{psk}_i)$
  (3) Record (tickets, sid, $(\mathsf{tikid}_i, \mathsf{psk}_i)_{i\in[N]}$)
  (4) Output (OK) to $\mathcal{Z}$
- Input (OnlinePhase, sid, ()) from $\mathcal{Z}$:
  (1) Lookup and remove (tickets, $(\mathsf{tikid}_i, \mathsf{psk}_i)_{i\in[N]}$); Abort if not found.
  (2) For $i \in [N]$, set $\mathsf{csecret}_i, \mathsf{cshare}_i, \mathsf{bkey}_i \leftarrow \mathsf{CHSecrets}(\mathsf{psk}_i)$
  (3) Send direct message $(\mathsf{StartHS}, \mathsf{sid}, (\mathsf{cshare}_i, \mathsf{bkey}_i, \mathsf{tikid}_i)_{i\in[N]})$ to $\mathsf{pid}_{\mathcal{P}}$
  (4) Continue upon receiving direct message (SKeyShares, sid, $(\mathsf{sshare}_i)_{i\in[N]}$) from $\mathsf{pid}_{\mathcal{P}}$
  (5) For $i \in [N]$, compute $\mathsf{hs_sec}_i, \mathsf{main_sec}_i \leftarrow \mathsf{MainSecrets}(\mathsf{psk}_i, \mathsf{csecret}_i, \mathsf{sshare}_i)$
  (6) Return (OK) to $\mathsf{pid}_{\mathcal{P}}$
  (7) Continue upon receiving message (EvalInitial, $\mathsf{sid}^{\mathsf{2PC}}$) from $\mathcal{F}_{\mathsf{2PC}}$
  (8) Return $((\mathsf{hs_sec}_i, \mathsf{main_sec}_i)_{i\in[N]})$ to $\mathcal{F}_{\mathsf{2PC}}$
  (9) Continue upon receiving message (EvalFinish, $\mathsf{sid}^{\mathsf{2PC}}$) from $\mathcal{F}_{\mathsf{2PC}}$
  (10) Return (OK) to $\mathcal{F}_{\mathsf{2PC}}$
  (11) Continue upon receiving direct message (Commitment, sid, com) from $\mathsf{pid}_{\mathcal{P}}$
  (12) Return $((\mathsf{csecret}_i, \mathsf{psk}_i)_{i\in[N]})$ to $\mathsf{pid}_{\mathcal{P}}$
  (13) Continue upon receiving message (Verified, $\mathsf{sid}^{\mathsf{2PC}}$, $([\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{V}})$) from $\mathcal{F}_{\mathsf{2PC}}$
  (14) Wait for message (ZKProof, $\mathsf{sid}^{\mathsf{ZK}}$, $(\mathsf{com}', [\![\mathsf{k}^C_{\mathsf{App}}]\!]'_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]'_{\mathcal{V}})$) from $\mathcal{F}_{\mathsf{ZK}}$
  (15) If $(\mathsf{com}, [\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{V}}) \neq (\mathsf{com}', [\![\mathsf{k}^C_{\mathsf{App}}]\!]'_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]'_{\mathcal{V}})$, Abort.
  (16) Output $(\mathsf{com}, [\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{V}})$ to $\mathcal{Z}$ and Halt

Figure 15: Verifier $\mathcal{V}$'s part of Protocol π_{TTH}

Protocol π_{TTH}, in the context of a Server, responds to the following messages:

- Message (TicketRequest, $\mathsf{sid}^{\mathsf{TLS}}$, psk, pid_C) from $\mathcal{F}_{\mathsf{TLS}}$:
  (1) Output (ChooseTicketId, sidTLS, pid_C) to $\mathcal{Z}$ and wait for response input (tikid)
  (2) Add ($\mathsf{sid}^{\mathsf{TLS}}$, tikid, psk) to initially-empty saved list tickets
  (3) Return (tikid) to $\mathcal{F}_{\mathsf{TLS}}$
- Message (ECHRequest, $\mathsf{sid}^{\mathsf{TLS}}$, pid_C) from $\mathcal{F}_{\mathsf{TLS}}$:
  (1) Output (GetECHConfig, sidTLS, pid_C) to $\mathcal{Z}$ and wait for response input (echconfig, echsecret)
  (2) Add ($\mathsf{sid}^{\mathsf{TLS}}$, echconfig, echsecret) to initially-empty saved list eches
  (3) Return (echconfig) to $\mathcal{F}_{\mathsf{TLS}}$
- Message (TCPMessage, sessionId, payload) from $\mathcal{F}_{\mathsf{TCP}}$:
  (1) Process payload as a normal TLS message according to the TLS 1.3 spec, using saved tickets and eches.
  (2) If payload decrypts to a plaintext req, output (ClientRequest, sessionId, req) to $\mathcal{Z}$ and wait for response input (resp).
  (3) Send (RecordReqResp, $\mathsf{sid}^{\mathsf{TLS}}$, req, resp) to $\mathcal{F}_{\mathsf{TLS}}$ and wait for response (OK)
  (4) Compute responsePayload with an encryption of plaintext resp according to the TLS 1.3 spec
  (5) Send (Transmit, sessionId, responsePayload) to $\mathcal{F}_{\mathsf{TCP}}$

Figure 16: Server's part of Protocol π_{TTH}

After the requests and responses. Later, after the second round of messages with the servers, the ideal functionality $\mathcal{F}_{\mathsf{MSO}}$ decrypts each request/reply with the real servers, and then computes a commitment to the selected request/reply with will be the eventual output to $\mathcal{V}$ if everything succeeds. Because $\mathsf{pid}_{\mathcal{V}}$ is corrupted, this commitment is leaked to Sim immediately, before the honest $\mathcal{P}$ sees their output.

From the way our protocol constructs the message commitment, this value contains exactly commitments to the request/reply ciphertexts, commitments to shares of the application traffic secrets, and (in the clear) the Verifier's share of the application traffic secrets.

These commitments are exactly the same as what the real prover sends to the verifier in the protocol, so Sim sends them to $\mathcal{A}$ at this time, after which $\mathcal{A}$ should reveal its original key share secrets α_i and ticket pre-shared keys psk_i. The simulator can now go back and check the validity and consistency of everything it was told earlier by $\mathcal{A}$, which Sim uses to then inform $\mathcal{F}_{\mathsf{MSO}}$ if the protocol should fail. If not, then the ideal functionality $\mathcal{F}_{\mathsf{MSO}}$ sends the honest prover its output, and Sim completes the protocol with $\mathcal{A}$. Sim sends to $\mathcal{A}$ the verifier's application traffic key share *as if* it is the final output from the 2PC, and sends $\top$ as if it is the output from $\mathcal{F}_{\mathsf{ZK}}$.

Notice that Sim learns absolutely nothing here that $\mathcal{A}$ did not already know or would not eventually know. It does not actually have to provide the commitment openings to the final ZK proof, because $\mathcal{F}_{\mathsf{ZK}}$ is a local subroutine which the simulator Sim is also responsible to handle.

F.2 Corrupted Prover

If $\mathcal{P}$ is corrupt but $\mathcal{V}$ is honest, then the protocol is kicked off by $\mathcal{V}$ asking the ideal functionality $\mathcal{F}_{\mathsf{MSO}}$ to start the online phase.

Then, because $\mathsf{pid}_{\mathcal{P}}$ is corrupt, the ideal functionality does not actually communicate with the real servers, but instead just leaks all of the tickets and corresponding PSKs (as provided by the honest Verifier) to $\mathcal{A}$.

Execution is now passed to Sim who, armed with the tickets and PSKs, must try to learn the commitment and decommitment for the corrupted prover's selected index.

The job of the simulator is relatively straightforward here: it simply acts exactly as an honest Verifier in the real protocol, including accurately and honestly simulating the 2PC.

At the end of the real interaction with $\mathcal{A}$, the simulator Sim is provided some purported commitment to the selected request/reply pair with one of the servers. In addition, because it is simulating the sub-functionalities, Sim also has the corrupted prover's values for its selected index, its plaintext query, and (crucially) its share of the application traffic secrets. From simulating $\mathcal{F}_{\mathsf{ZK}}$, Sim also learns

Simulator Sim for π_{TTH} responds to the following messages:

- Direct message $(\mathsf{StartHS}, \mathsf{sid}^{\mathsf{TTH}}, (\mathsf{cshare}_i, \mathsf{bkey}_i, \mathsf{tikid}_i)_{i\in[N]})$ to honest $\mathsf{pid}_{\mathcal{P}}$ from corrupted $\mathsf{pid}_{\mathcal{V}}$ via $\mathcal{A}$:
  (1) Send $(\textsc{OnlinePhase}, \mathsf{sid}^{\mathsf{MSO}}, (\mathsf{cshare}_i, \mathsf{bkey}_i, \mathsf{tikid}_i)_{i\in[N]})$ to $\mathcal{F}_{\mathsf{MSO}}$ on behalf of $\mathsf{pid}_{\mathcal{V}}$
  (2) Continue upon receiving $(\textsc{Round1Done}, \mathsf{sid}^{\mathsf{MSO}}, (\mathsf{svrMsg1}_i)_{i\in[N]})$ from $\mathcal{F}_{\mathsf{MSO}}$
  (3) Parse out each sshare_i from corresponding $\mathsf{svrMsg1}_i$
  (4) Send $(\textsc{SKeyShares}, \mathsf{sid}^{\mathsf{TTH}}, (\mathsf{sshare}_i)_{i\in[N]})$ to $\mathsf{pid}_{\mathcal{V}}$ from $\mathsf{pid}_{\mathcal{P}}$ via $\mathcal{A}$ and wait for response (OK)
  (5) Send $(\textsc{EvalInitial}, \mathsf{sid}^{\mathsf{2PC}})$ to $\mathsf{pid}_{\mathcal{V}}$ from simulated $\mathcal{F}_{\mathsf{2PC}}$ and wait for response $((\mathsf{hs_sec}_i, \mathsf{main_sec}_i)_{i\in[N]})$
  (6) Send $(\textsc{EvalFinish}, \mathsf{sid}^{\mathsf{2PC}})$ to $\mathsf{pid}_{\mathcal{V}}$ from simulated $\mathcal{F}_{\mathsf{2PC}}$ and wait for response (OK)
  (7) Return $((\mathsf{hs_sec}_i, \mathsf{main_sec}_i)_{i\in[N]})$ to $\mathcal{F}_{\mathsf{MSO}}$
  (8) Continue upon receiving $(\textsc{Round2Done}, \mathsf{sid}^{\mathsf{MSO}}, \mathsf{com}^{\mathsf{MSO}})$ from $\mathcal{F}_{\mathsf{MSO}}$
  (9) Parse $\mathsf{com}^{\mathsf{MSO}}$ as $(\mathsf{com}^{\mathsf{TTH}}, [\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{V}})$
  (10) Send $(\textsc{Commitment}, \mathsf{sid}^{\mathsf{TTH}}, \mathsf{com}^{\mathsf{TTH}})$ to $\mathsf{pid}_{\mathcal{V}}$ from $\mathsf{pid}_{\mathcal{P}}$ via $\mathcal{A}$ and wait for response $((\mathsf{csecret}_i, \mathsf{psk}_i)_{i\in[N]})$
  (11) Re-derive all previous values received from $\mathsf{pid}_{\mathcal{V}}$ via $\mathcal{A}$ using $\mathsf{csecret}_i$ and psk_i; if any do not match, then return $(\mathsf{shouldFail} = \top)$ to $\mathcal{F}_{\mathsf{MSO}}$ and do not continue.
  (12) Otherwise, return $(\mathsf{shouldFail} = \bot)$ to $\mathcal{F}_{\mathsf{MSO}}$
  (13) Continue upon receiving response $(\textsc{com})$ from $\mathcal{F}_{\mathsf{MSO}}$
  (14) Send $(\textsc{Verified}, \mathsf{sid}^{\mathsf{2PC}}, ([\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{V}}))$ to $\mathsf{pid}_{\mathcal{V}}$ from simulated $\mathcal{F}_{\mathsf{2PC}}$
  (15) Send $(\textsc{ZKProof}, \mathsf{sid}^{\mathsf{ZK}}, (\mathsf{com}^{\mathsf{TTH}}, [\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{V}})$ to $\mathsf{pid}_{\mathcal{V}}$ from simulated $\mathcal{F}_{\mathsf{ZK}}$
- Backdoor message $(\mathsf{GetReqResp}, \mathsf{sid}^{\mathsf{MSO}}, (\mathsf{tikid}_i, \mathsf{psk}_i)_{i\in[N]})$ from $\mathcal{F}_{\mathsf{MSO}}$:
  (1) For $i \in [N]$, set $\mathsf{csecret}_i, \mathsf{cshare}_i, \mathsf{bkey}_i \leftarrow \mathsf{CHSecrets}(\mathsf{psk}_i)$
  (2) Send $(\mathsf{StartHS}, \mathsf{sid}^{\mathsf{TTH}}, (\mathsf{cshare}_i, \mathsf{bkey}_i, \mathsf{tikid}_i)_{i\in[N]})$ to $\mathsf{pid}_{\mathcal{P}}$ from $\mathsf{pid}_{\mathcal{V}}$ via $\mathcal{A}$
  (3) Continue upon receiving $(\textsc{SKeyShares}, \mathsf{sid}^{\mathsf{TTH}}, (\mathsf{sshare}_i)_{i\in[N]})$ to $\mathsf{pid}_{\mathcal{V}}$ from $\mathsf{pid}_{\mathcal{P}}$ via $\mathcal{A}$
  (4) For $i \in [N]$, compute $\mathsf{hs_sec}_i, \mathsf{main_sec}_i \leftarrow \mathsf{MainSecrets}(\mathsf{psk}_i, \mathsf{csecret}_i, \mathsf{sshare}_i)$
  (5) Return (OK) to $\mathsf{pid}_{\mathcal{P}}$ via $\mathcal{A}$
  (6) Continue upon receiving message $(\textsc{Garble}, \mathsf{sid}^{\mathsf{TPC}}, (\mathsf{sel}, h_{1,\mathsf{sel}}, \mathsf{req}_{\mathsf{sel}}), (C^{\mathsf{fail}}, C^{\mathsf{leak}}))$ from $\mathsf{pid}_{\mathcal{P}}$ to simulated $\mathcal{F}_{\mathsf{2PC}}$ via $\mathcal{A}$
  (7) If $C^{\mathsf{fail}}((\mathsf{hs_sec}_i, \mathsf{main_sec}_i)_{i\in[N]}) = 1$, then Abort this execution.
  (8) Else If $C^{\mathsf{leak}}((\mathsf{hs_sec}_i, \mathsf{main_sec}_i)_{i\in[N]}) = 1$, then
    (a) Set $\mathsf{leaked} \leftarrow 1$
    (b) Return $(\textsc{Leaked}, (\mathsf{hs_sec}_i, \mathsf{main_sec}_i)_{i\in[N]})$ to $\mathsf{pid}_{\mathcal{P}}$ from simulated $\mathcal{F}_{\mathsf{2PC}}$
  (9) Else
    (a) Set $\mathsf{leaked} \leftarrow 0$
    (b) Compute $\mathsf{CHTS}_{\mathsf{sel}}, \mathsf{SHTS}_{\mathsf{sel}}$ from $\mathsf{hs_sec}_{\mathsf{sel}}$ and $h_{1,\mathsf{sel}}$
    (c) Return $(\mathsf{CHTS}_{\mathsf{sel}}, \mathsf{SHTS}_{\mathsf{sel}})$ to $\mathsf{pid}_{\mathcal{P}}$ from simulated $\mathcal{F}_{\mathsf{2PC}}$
  (10) Continue upon receiving $(\textsc{Commitment}, \mathsf{sid}, \mathsf{com}^{\mathsf{TTH}})$ to $\mathsf{pid}_{\mathcal{V}}$ from $\mathsf{pid}_{\mathcal{P}}$ via $\mathcal{A}$
  (11) Return $((\mathsf{csecret}_i, \mathsf{psk}_i)_{i\in[N]})$ to $\mathsf{pid}_{\mathcal{P}}$ via $\mathcal{A}$
  (12) Continue upon receiving message $(\textsc{Release}, \mathsf{sid}^{\mathsf{2PC}}, (\mathsf{hs_sec}'_i, \mathsf{main_sec}'_i)_{i\in[N]})$ from $\mathsf{pid}_{\mathcal{P}}$ to simulated $\mathcal{F}_{\mathsf{2PC}}$ via $\mathcal{A}$
  (13) Wait for message $(\textsc{ZKProve}, \mathsf{sid}^{\mathsf{ZK}}, (\mathsf{com}'^{\mathsf{TTH}}, [\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{V}}), \mathsf{decom})$ from $\mathsf{pid}_{\mathcal{P}}$ to simulated $\mathcal{F}_{\mathsf{ZK}}$ via $\mathcal{A}$
  (14) If any received values do not match the earlier ones, Abort this execution.
  (15) Set $\mathsf{com}^{\mathsf{MSO}} \leftarrow (\mathsf{com}^{\mathsf{TTH}}, [\![\mathsf{k}^C_{\mathsf{App}}]\!]_{\mathcal{V}}, [\![\mathsf{k}^S_{\mathsf{App}}]\!]_{\mathcal{V}})$
  (16) Return $(\mathsf{com}^{\mathsf{MSO}}, \mathsf{decom})$ to $\mathcal{F}_{\mathsf{MSO}}$

Figure 17: Simulator for π_{TTH}

the corrupted prover's decommitments to the selected query and response.

If all of these values are confirmed by Sim to be consistent with each other, then the simulator sends the commitment and decommitment back to the ideal functionality, which in turn will hand the commitment over to the true Verifier.

Crucially, if all of the servers are honest, then Sim knows from these checks that the committed request/response really must have come from one of the servers. The reason is that the application traffic keys are derived from some psk which only the simulator and (honest) server know, and the committed-to ciphertexts were provably obtained and decrypted before revealing any of the psk's to the adversary.