Janus: Fast Privacy-Preserving Data Provenance For TLS

Jan Lauinger
Technical University of Munich
Munich, Germany
jan.lauinger@tum.de

Jens Ernstberger
Technical University of Munich
Munich, Germany
jens.ernstberger@tum.de

Andreas Finkenzeller
Technical University of Munich
Munich, Germany
andreas.finkenzeller@tum.de

Sebastian Steinhorst
Technical University of Munich
Munich, Germany
sebastian.steinhorst@tum.de

Abstract

Web users can gather data from secure endpoints and demonstrate the provenance of sensitive data to any third party by using privacy-preserving TLS oracles. In practice, privacy-preserving TLS oracles remain limited and cannot verify larger, sensitive data sets. In this work, we introduce new optimizations for TLS oracles, which enhance the efficiency of selectively verifying the provenance of confidential web data. The novelty of our work is a construction which secures an honest verifier zero-knowledge proof system in the asymmetric privacy setting while retaining security against malicious adversaries. Concerning TLS 1.3 in the one round-trip time (1-RTT) mode, we propose a new, optimized garble-then-prove paradigm in a security setting with malicious adversaries. Our improvements reach new performance benchmarks and facilitate a practical deployment of privacy-preserving TLS oracles in web browsers.

Keywords

Data Provenance, Transport Layer Security, Zero-knowledge Proof, Secure Two-party Computation, TLS Oracle

1 Introduction

In the current age of the Internet where generative artificial intelligence (AI) boosts the spread of misinformation as never before, industry leading companies combat misinformation with new data provenance initiatives to maintain a responsible and verifiable data economy [36, 46]. The goal of the initiatives is the establishment and integration of data provenance solutions into today's web, which lacks support of verifiable data provenance. For instance, secure channel protocols such as transport layer security (TLS) provide confidential and authenticated communication sessions between two parties: a client and a server. However, if clients present data of a TLS session to any third party (e.g. website), then the third party cannot verify if the presented data originated from an *authentic* and *correct* TLS session (cf. top part of Figure 1). Thus, the third party cannot verify the provenance of the TLS data. In the eyes of the third party, TLS data counts as *authentic* if the origin of the data can be verified. Further, TLS data counts as *correct* if the third-party is able to verify the integrity of presented TLS data against a valid TLS session.

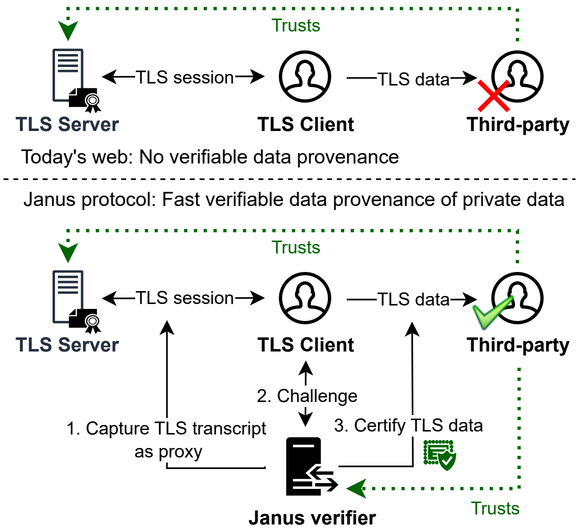


Figure 1: Illustration of TLS sessions in today's web (top part) and TLS sessions accompanied by a TLS oracle (bottom part). TLS sessions, per default, are secure channels between two parties and prevent a third-party from verifying the provenance of TLS data. In contrast, TLS oracles use a trusted verifier to audit and certify the provenance of TLS data, making TLS data publicly verifiable.

To save a third party from individually verifying data provenance, current approaches either require *servers* to attest to TLS data via digital signatures [9, 45], or employ TLS oracles [43, 59]. Data attestation through *servers* is an efficient data provenance solution but requires server-side software changes and access to a certification infrastructure. By contrast, TLS oracles relieve *servers* from maintaining a data provenance infrastructure by taking over the provisioning and verification of data provenance. Due to the seamless integration into the web, TLS oracles count as *legacy-compatible* as they do not introduce any server-side changes. TLS oracles depend on a *verifier* to examine the provenance of TLS data (cf. Janus verifier at the bottom of Figure 1). To validate the provenance of TLS data, the *verifier* captures the transcript of a

Proceedings on Privacy Enhancing Technologies 2025(1), 511–530
© 2025 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2025-0028

TLS session and challenges the TLS client with a proof computation. If a TLS client can prove *authenticity* and *correctness* of secret TLS session parameters against the captured TLS transcript at the *verifier*, then the *verifier* certifies the TLS data of the client. With the certificate, TLS clients are able to convince any third party of data provenance if the third party trusts the *verifier*.

TLS oracles have originated in the context of blockchain ecosystems, where TLS oracles originally solved the "oracle problem" of importing trustworthy data feeds to isolated smart contracts [43, 58, 59]. However, TLS oracles are generally applicable to the Internet, which makes them a crucial technique to build user-centric and data-sovereign systems [17]. For instance, through TLS oracles, users are able to present solvency checks without giving up control and privacy of their data [38]. The accountability and credibility guarantees of data provenance systems are used to combat price discrimination [59], bootstrap legacy credentials [39], or attest if a digital resource originated from a generative AI website [4].

Challenge: Even though different solutions exist, TLS oracles remain constrained in the amount of sensitive data they can validate. This means that for larger sensitive resources such as confidential documents, images, or data sets, data provenance solutions are impractical. For instance, clients are required to prove non-algebraic encryption algorithms (e.g. AES128) in zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) proof systems [59]. However, current zkSNARK proof systems operate efficiently if the computed algorithm relies on algebraic structures (e.g. MiMC [3]). Current approaches leverage the structure of TLS 1.3 stream ciphers and separate non-algebraic algorithms from the computations performed by the zkSNARK proof system [43]. In this case, the *client* is required to know the structure of TLS data in advance and cannot selectively verify dedicated parts of TLS records. Even if the computation of non-algebraic algorithms is shifted into a pre-computation phase [57], end-to-end (E2E) efficiency of private provenance solutions remains expensive.

Contribution: Our work addresses the above mentioned limitations with two new contributions. We leverage the fact that, in the challenge phase (cf. stage 2 in Figure 1), TLS oracles introduce an asymmetric privacy setting between collaboratively acting parties; the TLS *client* and the *verifier*. We leverage the asymmetric privacy setting to construct a honest verifier zero-knowledge (HVZK) proof system with security against malicious adversaries. Our construction relies on a new validation phase which is unilaterally performed by the *client*. The benefit of the HVZK proof system [29] is that it efficiently evaluates non-algebraic algorithms and improves proof computations in the challenge phase. Our approach does not require a trusted setup security assumption. With that, our work achieves new E2E benchmarks and solves a main bottleneck of current TLS oracles; the efficient evaluation of legacy algorithms without compromising on security guarantees. Our first contribution applies to TLS oracles running TLS 1.2 or 1.3.

Our second contribution applies to TLS 1.3 in the one round-trip time (1-RTT) mode. Here, we require the *client* to select a cipher suite which is supported by the *server*. In a non-optimistic scenario, the *client* is supposed to perform one pre-fetch call. If the *client* sends a compliant client hello (CH) message during the TLS 1.3 handshake, then the *server* instantly responds with the entire *server*-side handshake transcript. We leverage this effect and show that the *verifier* can securely authenticate the server handshake traffic secret (SHTS) in a malicious security setting. With access to an authentic SHTS at the *verifier*, we run the garble-then-prove paradigm [53] and rely on a semi-honest two party computation (2PC) system which does not depend on authenticated garbling. We detect malicious activities of a *client* by matching transcript commitments against authenticity guarantees derived from SHTS. We achieve performance advantages by utilising more lightweight, semi-honest 2PC for the majority of TLS 1.3 computations.

Result: Our E2E benchmarks for TLS 1.3 verify 8 kB of public TLS data in 0.58 seconds and verify 8 kB of sensitive TLS data in 6.7 seconds. Running TLS 1.2, we verify 8 kB of sensitive TLS data in 6.2 seconds. Concerning proof computations in the client challenge, our work outperforms related approaches by a factor of 8x (cf. Section 6.2) and relies on a security setting which does not require a trusted setup assumption. In analogy to Roman mythology, we name our contributions for TLS oracles after the god of transitions, *Janus*. With that, the *Janus* optimizations guard an efficient transition of web resources into a representation where provenance can be verified. In summary,

- We formalize the asymmetric privacy setting of TLS 1.2 and TLS 1.3 oracles. We show that in the asymmetric privacy setting, maliciously secure proof systems can be replaced with a construction that combines a HVZK proof system and a new unilateral validation phase.
- We optimize the efficiency of TLS 1.3 oracles by considering SHTS authenticity guarantees during the garble-then-prove paradigm while retaining security properties equivalent to previous works.
- We analyse the security of our constructions (cf. Appendix B), provide performance benchmarks (cf. Section 6), and open-source[1] the implementation of our secure computation building blocks.

2 Preliminaries

This section highlights the key concepts of TLS which data provenance solutions build upon. Extensive details of each cryptographic construction or protocol are in the Appendix A.

2.1 General Notations

The TLS notations of this work are introduced in Section 2.2, and closely follow the notations of the work [16]. Further, we denote vectors as bold characters $\mathbf{x} = [x_1, \ldots, x_n]$, where $len(\mathbf{x}) = n$ returns the length of the vector. Base points of elliptic curves are represented by $G \in EC(\mathbb{F}_p)$, where the finite field $\mathbb{F}$ is of a prime size p. For elliptic curve elements, the operators $\cdot, +$ refer to the scalar multiplication and addition of elliptic curve points $P \in EC(\mathbb{F}_p)$. The symbol λ indicates the security parameter. For bits or bit strings, the operators $\cdot$ represents the logical AND, and $\oplus$ represents the logical XOR. Other operators describe a random assignment of a variable with $\overset{\$}{\leftarrow}$, the concatenation of strings with $||$, and the comparison of variables with $\overset{?}{=}$.

[1] https://github.com/jplaui/circuits_janus

Table 1: Notations and formulas of TLS variables.

Variable	Formula
H_2	H(ClientHello‖ServerHello)
H_3	H(ClientHello‖…‖ServerFinished)
H_6	H(ClientHello‖…‖ServerCert)
H_7	H(ClientHello‖…‖ServerCertVfy)
H_9	H(ClientHello‖…‖ClientCertVfy)
label_{11}	"TLS 1.3, server CertificateVerify"
(k_{SATS}, iv_{SATS}) | (k_{CATS}, iv_{CATS})	DeriveTK(s=SATS|CATS) = (**hkdf.exp**(s,"key",H(" "),len(k)), **hkdf.exp**(s,"iv",H(" "),len(iv)))

2.2 Transport Layer Security

TLS is a standardized suite of cryptographic algorithms to establish secure and authenticated communication channels between two parties. TLS exists in different versions; TLS 1.2 and TLS 1.3. TLS has two phases, where the *handshake phase* derives cryptographic parameters to secure data sent in the *record phase*. TLS relies on the algorithms of hash-based message authentication code (HMAC) and HMAC-based key derivation function (HKDF) to securely derive cryptographic parameters and relies on digital signatures to authenticate parties (cf. **ds.Sign**, **ds.Verify**, **hkdf.ext**, **hkdf.exp**, **hmac** in Figure 2). We provide further details of TLS-specific security algorithms in the Appendix A and present TLS-specific transcript hashes, labels, and key derivation functions of traffic keys in Table 1.

2.2.1 Handshake Phase. **Key Exchange and Key Derivation:** To establish a secure channel between a server and a client, TLS relies on the Diffie-Hellman key exchange (DHKE) to securely exchange cryptographic secrets between two parties (cf. Figure 2, lines 1-4). For example, with TLS 1.3 configured to use elliptic curve cryptography, parties protect secrets x, y with an encrypted representation X, Y and exchange X, Y via the CH and server hello (SH) messages m_{CH}, m_{SH}. With access to X, Y, only the client and server can securely derive the Diffie–Hellman ephemeral (DHE) key, where DHE $= x \cdot y \cdot G = y \cdot X = x \cdot Y$ holds. Both parties continue to use DHE to derive traffic secrets. In the TLS 1.3 1-RTT mode, the *server* is able to encrypt all server-side handshake messages after receiving a supported client key share in the CH message m_{CH}.

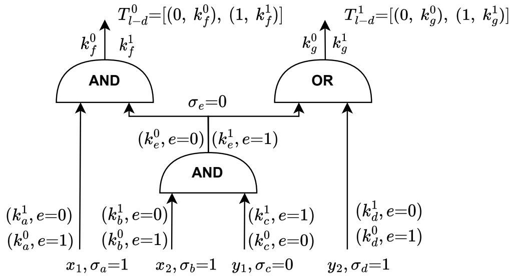
In contrast, TLS 1.2 exchanges the messages m_{CH}, m_{SH} in plain and refers to the DHE value as the premaster secret. TLS 1.2 uses the premaster secret together with the client and server randomness to derive a master secret, which, in turn, is used to derive traffic secrets. When TLS 1.2 is configured to use authenticated encryption with associated data (AEAD) based on stream ciphers, TLS 1.2 generates two application traffic keys to secure record phase traffic (k_{CATS}, k_{SATS}). Otherwise, if TLS 1.2 uses a cipher block chaining (CBC) mode to encrypt records, TLS 1.2 generates additional message authentication code (MAC) keys. In contrast to the Galois Counter Mode (GCM) mode, the CBC mode counts as key-committing [20, 57], which guarantees the existence of a non-ambiguous mapping between traffic secrets and authentication tags. Per default, TLS 1.3 generates two keys (k_{CHTS}, k_{SHTS}) to secure handshake traffic and generates two keys to secure record traffic (k_{SATS}, k_{CATS}). Due to the key-independence property of TLS 1.3 [15], disclosing handshake traffic secrets (e.g. SHTS) does not compromise the security of record traffic secrets. For instance, to compute the server application traffic secret (SATS), a party requires access to the handshake secret (HS). Even though, HS is used to derive SHTS (cf. line 5 of Figure 2), **hkdf.exp** prevents the reconstruction of HS from SHTS.

```
TLS Handshake between the client c and server s:
inputs: $x \xleftarrow{\$} \mathbb{F}_p$ by c. ($y \xleftarrow{\$} \mathbb{F}_p, sk_S, pk_S$) by s.
outputs: ($tk_{\text{CATS}}, iv_{\text{CATS}}, tk_{\text{SATS}}, iv_{\text{SATS}}$) to c and s.
1. c: $X = x \cdot G$; send $X$ in $m_{\text{CH}}$
2. s: $Y = y \cdot G$; send $Y$ in $m_{\text{SH}}$
3. b: dES = hkdf.exp(hkdf.ext(0,0),"derived" || H(" "))
4. b: DHE = $x \cdot y \cdot G$; HS = hkdf.ext(dES, DHE)
5. b: SHTS = hkdf.exp(HS,"s hs traffic" || $H_2$)
6. b: CHTS = hkdf.exp(HS,"c hs traffic" || $H_2$)
7. b: ($k_{\text{CHTS}}, iv_{\text{CHTS}}$) = DeriveTK(CHTS)
8. b: ($k_{\text{SHTS}}, iv_{\text{SHTS}}$) = DeriveTK(SHTS)
9. b: $fk_S$ = hkdf.exp(SHTS, "finished" || " ")
10. s: SCV=ds.Sign($sk_S$,$\text{label}_{11}||H_6$); send SCV in $m_{\text{SCV}}$
11. s: SF = hmac($fk_S$, $H_7$); send SF in $m_{\text{SF}}$
12. c: SF' = hmac($fk_S$, $H_7$); verify SF' $\overset{?}{=}$ SF
13. c: ds.Verify($pk_S$, $\text{label}_{11}$ || $H_6$, SCV) $\overset{?}{=}$ 1
14. b: $fk_C$ = hkdf.exp(CHTS, "finished" || " ")
15. c: CF = hmac($fk_C$, $H_9$); send CF in $m_{\text{CF}}$
16. s: CF' = hmac($fk_C$, $H_9$); verify CF' $\overset{?}{=}$ CF
17. b: dHS = hkdf.exp(HS,"derived" || H(" "))
18. b: MS = hkdf.ext(dHS, 0)
19. b: CATS = hkdf.exp(MS, "c ap traffic" || $H_3$)
20. b: SATS = hkdf.exp(MS, "s ap traffic" || $H_3$)
21. b: ($k_{\text{CATS}}, iv_{\text{CATS}}$) = DeriveTK(CATS)
22. b: ($k_{\text{SATS}}, iv_{\text{SATS}}$) = DeriveTK(SATS)
```

Figure 2: TLS 1.3 specification of session secrets and keys. Characters at the beginning of lines indicate if the server s, the client c, or both parties b call the functions per line.

Authenticity: To mutually authenticate each other, both parties exchange certificates and compute authentication parameters (cf. Figure 2, lines 9-16). Notice that in TLS, client-side authentication is optional, which is why we omit client certificates in Figure 2. But, we show the computations of the server finished (SF) and client finished (CF) values, because, to constitute an authenticated TLS session, both parties must successfully exchange and verify the SF and CF messages $m_{\text{SF}}, m_{\text{CF}}$. For a server-side authentication, the server computes the certificate verification value (e.g. SCV), which binds a Public Key Infrastructure (PKI) X.509 certificate to the TLS transcript via a digital signature [1]. Here, the signature is computed with the server secret key sk_S and is verified with the corresponding server public key pk_S. The client obtains the server public key pk_S in the PKI certificate and aborts the TLS session if the signature verification fails.

2.2.2 Record Phase. The TLS record phase requires parties to protect data with authenticated encryption (AE) algorithms before data can be exchanged. AE algorithms translate plaintext data **pt** into a



$G(\mathcal{C}^{0,(1,2)}_{\text{AND}})$	
0,0	$\text{E}_{k^1_b}(\text{E}_{k^0_c}(k^0_e\|\|0))$
0,1	$\text{E}_{k^1_b}(\text{E}_{k^1_c}(k^1_e\|\|1))$
1,0	$\text{E}_{k^0_b}(\text{E}_{k^0_c}(k^0_e\|\|0))$
1,1	$\text{E}_{k^0_b}(\text{E}_{k^1_c}(k^0_e\|\|0))$

$G(\mathcal{C}^{1,(0,1)}_{\text{AND}})$	
0,0	$\text{E}_{k^1_a}(\text{E}_{k^0_e}(k^0_f))$
0,1	$\text{E}_{k^1_a}(\text{E}_{k^1_e}(k^1_f))$
1,0	$\text{E}_{k^0_a}(\text{E}_{k^0_e}(k^0_f))$
1,1	$\text{E}_{k^0_a}(\text{E}_{k^1_e}(k^0_f))$

$G(\mathcal{C}^{1,(1,3)}_{\text{OR}})$	
0,0	$\text{E}_{k^0_e}(\text{E}_{k^1_d}(k^1_g))$
0,1	$\text{E}_{k^0_e}(\text{E}_{k^0_d}(k^0_g))$
1,0	$\text{E}_{k^1_e}(\text{E}_{k^1_d}(k^1_g))$
1,1	$\text{E}_{k^1_e}(\text{E}_{k^0_d}(k^1_g))$

Figure 3: Example of a garbled circuit C expressing the function f of a secure computation via boolean logic gates. Every circuit wire w_L is encoded with secret internal labels i, a secret and random signal bit σ_L, external labels e=$\sigma_L \oplus$ i (where $i, e, \sigma \in \{0,1\}$), and wire keys $\mathbf{k}^{\mathbf{i}}_L$. Internal labels are associated with input data bits and the lists T_{l-d} map output labels to output data bits.

confidential and authenticated representation ($\mathbf{ct}, t$), with ciphertext **ct** and authentication tag t. Ciphertext data is computed based on block or stream cipher algorithms and depends on keys established in the handshake phase. We elaborate on TLS data protection algorithms in the Appendix A.4.

2.3 Cryptographic Building Blocks

This section provides an overview of the cryptographic fundamentals that support the *Janus* optimizations. Formal descriptions of cryptographic building blocks can be found in the Appendix A.

2.3.1 Semi-honest 2PC with Garbled Circuits (GCs). Secure 2PC allows two mutually distrusting parties with private inputs x, y to jointly compute a public function $f(x, y)$ without learning the counterparty's private input [34, 55]. A 2PC system based on boolean garbled circuits involves a party p_1 with input **x** as the garbler and party p_2 with input **y** as the evaluator. Party p_1 is supposed to generate the garbled circuit $G(C)$, where the boolean circuit C implements the logic of the public function f (cf. Figure 3). To generate the garbled circuit, p_1 randomly samples wire keys $\mathbf{k}^0_L, \mathbf{k}^1_L$ and a signal bit σ_L at every wire w_L. For the purpose of evaluating the function f, wire keys $\mathbf{k}^{\mathbf{i}}_L$ encode binary data representations of f using internal labels **i**. The purpose of signal bits is twofold. Signal bits encrypt internal bits to external bits e_L=$\sigma \oplus$ **i** which can be shared with p_2. With that, signal bits enable the evaluator to discover valid entries of garbled tables $G(C)$ through external bits e [26]. Further, signal bits randomize garbled truth tables $G(C)$ to obfuscate truth table bit mappings.

Once wire keys, signal bits, and external labels exist, p_1 computes the garbled table entries as follows. Per row of table $G(C)$ (cf. Figure 3), the bit tuples in the left column are combinations of external labels which correspond to incoming gate wires. The right column contains double encrypted wire keys that correspond to outgoing gate wires. For gates yielding output labels, garbled entries encrypt wire keys. For intermediate gates, garbled entries encrypt wire keys concatenated with corresponding external labels.

After garbling a circuit, p_1 shares $\boldsymbol{G(C)}$, $\boldsymbol{T_{l-d}}$, and, if **x**=[1,0], (k^1_a, e=0) and (k^0_b, e=1) with p_2. To obtain wire keys that correspond to the input bits of **y**, p_2 interacts with p_1 in two 1-out-of-2 Oblivious Transfer (OT) protocols (cf. Section 2.3.2). The OT protocol requires p_1 to share $k^{\mathbf{y}}_e, k^{\mathbf{y}}_d$ with corresponding external values with p_2. Further, the OT scheme gives p_2 access to the keys (k^0_c, e=0) and (k^1_d, e=0) if **y**=[0,1], and prevents p_1 from learning p_2's selection of wire keys. With access to $\boldsymbol{G(C)}$, input wire keys and corresponding external labels, p_2 is able to evaluate the garbled circuit. To evaluate the first output bit, p_2 decrypts the third entry of table $G(C^{0,(1,2)}_{AND})$ and obtains (k^0_e, e=0). With that, p_2 continues to decrypt the first entry of table $G(C^{1,(0,1)}_{AND})$ to obtain k^0_f (cf. Figure 3). Last, p_2 decodes k^0_f using the decoding table T^0_{l-d} to obtain the first output bit 0. If required, p_2 shares the obtained 2PC output back to p_1.

2.3.2 Oblivious Transfer. Secure 2PC based on GCs depends on the 1-out-of-2 OT^1_2 sub protocol to secretly exchange input parameters of the circuit [11]. The OT^1_2 scheme involves two parties where party p_1 sends two messages m_1, m_2 to party p_2. Party p_1 does not learn which of the two messages m_b is revealed to party p_2. Party p_2 inputs a secret bit b which decides the selection of the message m_b. In this work, we make use of the OT^1_2 scheme defined in the work [11], which does not require a trusted setup. The trusted setup procedure introduces a third party which (i) takes over the generation of cryptographic material and (ii) is trusted to delete the underlying random parameters of the material.

2.3.3 2PC with Malicious Adversaries. We consider the work [27] to secure the semi-honest 2PC defined in Section 2.3.1 against malicious adversaries. The dual-execution mode in [27] runs two instances of the semi-honest 2PC, where both parties p_1 and p_2 successively act as the garbler and evaluator. Subsequently, both parties interact in a secure validation phase to verify if both executions yield the same output. We describe details of the maliciously secure 2PC system in the Appendix A.5.5.

2.3.4 Zero-knowledge Proofs based on Garbled Circuits. Proof systems allow a prover p to convince a verifier v of whether or not a statement is true. In theory, proof systems rely on a NP language $\mathcal{L}$ and the existence of an algorithm $R_{\mathcal{L}}$, which decides in polynomial time if witness w is a valid proof for the statement $x \in \mathcal{L}$ by evaluating $R_{\mathcal{L}}(x, w) \overset{?}{=} 1$. The assumption is that for any statement $x \in \mathcal{L}$, there exist a valid witness w and no witness exists for statements $x \notin \mathcal{L}$ [41, 50]. Proof systems provide the properties, where

Completeness ensures that an honest prover convinces an honest verifier by presenting a valid witness for a statement.
Soundness guarantees that a cheating prover cannot convince a honest verifier by presenting an invalid witness for a statement.
Zero-knowledge guarantees that a malicious verifier does not learn anything except the validity of the statement.
HVZK holds if the zero-knowledge property can be shown for a semi-honest verifier, who honestly follows the protocol definition.

Interestingly, zero-knowledge is a subset of secure 2PC and a zero-knowledge proof (ZKP) can be computed using GC-based 2PC if only one party inputs private data. In this work, we make use of the HVZK notion based on boolean GCs [29]. In this setting, the garbler and constructor of the GC acts as the verifier and is assumed to behave semi-honest. The GC evaluates a function f, which yields $\{0, 1\}$. The evaluator, as the prover, obtains the GC, input wire keys and corresponding external labels but does not obtain the decoding table. After the prover evaluates the GC and returns the wire key which corresponds to a 1, the verifier is convinced of the proof. Formal security proofs for *completeness*, *soundness*, and HVZK of the garbled circuits proof system are provided in the work [29].

2.3.5 Cryptographic Commitments. Commitment schemes allow a party to hide a message string m via a commitment string c. The **c.Commit** algorithm outputting c takes as input the message m and a secret commitment randomness r. To verify if m computes to c under randomness r requires a party to call an opening algorithm **c.Open**, which takes as input m, c, r. Commitments count as binding if a non-ambiguous mapping between m, r towards c exist.

Commitments are often used in protocols which rely on ZKP cryptography. Using a ZKP to compute the **c.Open** function allows a prover to convince the verifier from knowing a valid commitment opening without revealing the witness. We formally define commitment schemes in the Appendix A.8

2.3.6 Secret Sharing. Secret sharing involves a trusted dealer to break a secret into shares with a **ss.Share** algorithm. Shares are distributed to qualified recipients which can reconstruct the secret by computing individual shares back together with a **ss.Reconstruct** algorithm [24]. In this work, we consider secret sharing with an access structure of t=n=2, where t out of n parties must add together secret shares to reconstruct the secret [47]. We provide the formal definition of secret sharing in the Appendix A.7.

3 System Model

The system model defines system roles, the threat model, and system goals in form of security properties.

3.1 System Roles & Adversarial Behavior

Clients establish a TLS session with *servers*, query data from *servers*, and present TLS data proofs to the *verifiers*. We assume that *clients* behave maliciously and arbitrarily deviate from the protocol specification in order to learn secret shares of TLS parameters. Further, malicious *clients* try to learn any information that contributes to convincing the *verifier* of false statements on presented TLS data.
Servers participate in TLS sessions with *clients* and return record data upon the reception of compliant API queries. We assume honest *servers* which follow the protocol specification.
Verifiers act as proxies and take over the role of TLS oracle verifiers. *Verifiers* are configured at the client and route TLS traffic between the *client* and the *server*. We assume malicious *verifiers* deviating from the protocol specification with the goal to learn TLS session secret shares or private session data of *clients*.

3.2 Threat Model

We rely on a threat model with secure TLS communication channels between *clients* and *servers* (TLS security guarantees hold). Further, we assume that fresh randomness is used per TLS session. Network traffic, even if it is intercepted via a machine-in-the-middle (MITM) attack by the adversary (e.g. the *client*), cannot be blocked indefinitely. We assume up-to-date Domain Name System (DNS) records at the *verifier* such that the *verifier* can resolve and connect to correct Internet Protocol (IP) addresses of *servers*. The IP address of a *server* cannot be compromised by the adversary such that adversaries cannot request malicious PKI certificates for a valid DNS mapping between a domain and a *server* IP address. *Servers* share valid PKI certificates for the authenticity verification in the TLS handshake phase. Server impersonation attacks are infeasible because secret keys, which correspond to exchanged PKI certificates, are never leaked to adversaries. Our protocol imposes multiple verification checks on the *client* and the *verifier*, where failing verification leads to protocol aborts at the respective parties. All system roles are computationally bounded and learn message sizes of TLS transcript data. For employed ZKP systems, we expect completeness, soundness, and HVZK to hold. We assume that the *client* and *verifier* do not collude.

3.3 System Goals

The following security properties concern the *client* and *verifier* as the *server* is assumed to behave honestly.

Session-authenticity guarantees that *verifiers* attest web traffic which originates from an authentic TLS session. Authenticity is guaranteed if the *verifier* successfully verifies the PKI certificate of the server.
Session-integrity guarantees that a malicious *client* and *verifier* cannot deviate from the TLS specification if a TLS session has been authenticated. This means that an adversary cannot modify server-side or client-side TLS traffic in any TLS phase. Notice that for client-side TLS traffic of the record phase, a malicious *client* is able to send arbitrary queries to the *server*, such that *servers* decide if queries conform with API handlers.
Session-confidentiality guarantees that the *verifier* neither learns any entire TLS session secrets nor any record data which has been exchanged between the *client* and the *server*. Further, the notion guarantees that the *verifier* learns nothing beyond the fact that a statement on TLS record data is true or false.
MITM-resistance guarantees that the properties of *session-integrity*, *session-authenticity*, and *session-confidentiality* hold in a system setting, where adversaries are capable of mounting MITM attacks.

4 Optimizing Proof Computations In The Asymmetric Privacy Setting

We analyze TLS oracles with regard to performance tradeoffs and the asymmetric privacy setting (cf. Section 4.1). Next, we deploy a HVZK proof system in the asymmetric privacy setting (cf. Section 4.2). Our construction relies on a new secure validation phase to establish security against malicious adversaries (cf. Appendix B.1).

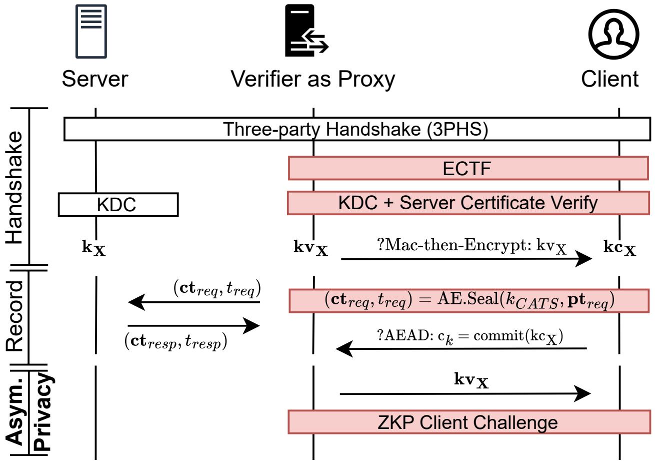

Server
Verifier as Proxy
Client
Handshake
Record
Asym. Privacy
Three-party Handshake (3PHS)
ECTF
KDC
KDC + Server Certificate Verify
$\mathbf{k_X}$
$\mathbf{kv_X}$
?Mac-then-Encrypt: $\mathrm{kv_X}$
$\mathbf{kc_X}$
$(\mathbf{ct}_{req}, t_{req})$
$(\mathbf{ct}_{req}, t_{req}) = \mathrm{AE.Seal}(k_{CATS}, \mathbf{pt}_{req})$
$(\mathbf{ct}_{resp}, t_{resp})$
?AEAD: $\mathrm{c}_k = \mathrm{commit(kc_X)}$
$\mathbf{kv_X}$
ZKP Client Challenge

Figure 4: High-level protocol of TLS oracles. After the key derivation computation (KDC), it holds that $k_X = kv_X + kc_X$, where X indicates server or client side AE keys. Algorithms executed by two parties are surrounded by red boxes and achieve security against malicious adversaries. The syntax '?encryption_mode:' on arrows applies the arrow if the TLS oracle configured the questioned encryption mode.

4.1 Analyzing Oracles & Asymmetric Privacy

In this section, we analyze the performance bottlenecks of TLS oracles and identify the conditions of asymmetric privacy.

4.1.1 Three-party Handshake. TLS oracles turn the two-party protocol of TLS into a three-party protocol by introducing a *verifier* [43]. The *verifier* ensures that the TLS data of the *client* preserves integrity according to an authenticated TLS session via a verifiable computation trace. To audit the integrity of TLS data, the *verifier* and *client* establish a mutually vetting but collaborative TLS client. To construct a collaborative TLS client, TLS oracles replace the TLS handshake with a three-party handshake (3PHS) [43, 59]. In the 3PHS, every party injects secret randomness such that the DHE secret on the client-side depends on two secrets. As such, the DHE value, which is individually derived at the *server*, can be jointly reconstructed if the *client* and *verifier* add shared secrets together. Appendix A.1 presents the cryptography of the 3PHS.

The consequence of the 3PHS is that the *client* depends on the computational interaction with the *verifier* to proceed in a TLS session with the *server*. The *client* preserves computational integrity according to the TLS specification if the joint TLS computations with the *verifier* progress. Without access to the secret share of the *verifier*, *clients* cannot derive and use full TLS secrets and encryption keys that are required for the secure session with the *server*. Introducing false session parameters on the client side leads to a TLS session abort at the server.

4.1.2 Client-side Two-party Computation. With secret shared TLS parameters, the *client* and *verifier* proceed according to the TLS specification by using secure 2PC techniques [10, 59]. To achieve efficient secure 2PC [43, 59], TLS oracles convert secret-shared DHE values in form of elliptic curve (EC) coordinates into bitwise additive secret shares with the Elliptic Curve to Field (ECTF) algorithm (cf. Appendix A.5) [19]. Additive secret shares can be efficiently added together in 2PC circuits that are based on boolean GCs [12, 27, 52, 55, 59]. After the ECTF conversion (cf. Figure 4), the *client* and *verifier* perform the TLS key derivation and record phase computations using maliciously secure 2PC based on boolean GCs, which comes with optimized binary circuits for the required computations (e.g. AES) [24, 25].

Mac-then-Encrypt (e.g. CBC-HMAC). The efficiency of TLS oracles in the record phase heavily depends on the cipher suite configuration. If TLS uses Mac-then-Encrypt (MtE) AE (TLS 1.2 with cipher block chaining hash-based message authentication code (CBC-HMAC)), then the *client* and *verifier* end up deriving four secret-shared keys in the handshake phase:

- k_{CATS}=kv_{CATS}+kc_{CATS} to encrypt request data $\mathbf{pt}_{req}$.
- k_{SATS}=kv_{SATS}+kc_{SATS} to encrypt response data $\mathbf{pt}_{resp}$.
- k^t_{CATS}=kv^t_{CATS}+kc^t_{CATS} to authenticate requests $\mathbf{t}_{req}$.
- k^t_{SATS}=kv^t_{SATS}+kc^t_{SATS} to authenticate responses $\mathbf{t}_{resp}$.

The *verifier* can disclose the encryption keys kv_{SATS}, kv_{CATS} to the *client*. However, key shares to compute authentication tags must be kept private in order to control the integrity of joint TLS computations [59]. Partly disclosing key shares improves the efficiency of 2PC computations in the record phase. Because, *clients* can encrypt or decrypt records locally without costly 2PC calls. Concerning the computation of MtE authentication tags, the 2PC complexity remains independent of record sizes (e.g. using a 2PC-optimized evaluation of HMAC) [59].

Another benefit of CBC-HMAC is that it counts as key committing [20, 59], which guarantees the existence of an unambiguous mapping between a TLS session key and record data. As a consequence, capturing **ct** is the only requirement for the *verifier* before secret shares can be disclosed to the *client*. TLS oracles use the key committing property and simplify the ZKP computation during the client challenge to (i) three invocations of advanced encryption standard (AES) and (ii) a selective data opening which leverages the Merkle–Damgård construction [18, 59].

AEAD (e.g GCM / CHACHA20_POLY1305). If TLS is configured to protect records with AEAD algorithms (TLS 1.3 and optionally TLS 1.2), then the *client* and *verifier* derive two secret-shared keys (k_{CATS}=kv_{CATS}+kc_{CATS}, k_{SATS}=kv_{SATS}+kc_{SATS}). Thus, to maintain *session-integrity*, the *verifier* cannot disclose any secret shared AEAD key in the record phase before receiving a commitment. Since keys are not decoupled as with CBC-HMAC, the efficiency of TLS oracles running AEAD cipher suites deteriorates for larger record sizes. The bottleneck is the 2PC computation of authentication tags, which evaluate algebraic structures over all ciphertext chunks (e.g. polynomials over large fields GF(2^{128}) for GCM and GF($2^{130} - 5$) for POLY1305).

Insight 1: The performance of 2PC AEAD tag computations deteriorates for larger record sizes.

Further, AEAD configurations require special attention as AEAD cipher suites are not key committing [21–23, 37, 57]. This means that an adversary can perform commitment attacks [40]. For example, the message franking attack finds two messages $m_1 \neq m_2$

```
$C_{\text{MtE}}(\mathbf{pt}, kc, kc^t \; ; \; kv, kv^t, \mathbf{ct}_\alpha{=}\mathbf{ct}[\text{end-3:}], \phi)$:
1. $\text{t'} = \text{HMAC}(kc^t + kv^t, \mathbf{pt})$
2. $\mathbf{ct}_\alpha\text{'} = \text{AES}(kc + kv, \text{t'})$
3. assert: $\mathbf{ct}_\alpha \overset{?}{=} \mathbf{ct}_\alpha\text{'}, 1 \overset{?}{=} f_\phi(\mathbf{pt})$
$C_{\text{AEAD}}(\mathbf{pt}, kc \; ; \; kv, \mathbf{ct}, \text{iv}, t_\alpha, \text{c}_k, \phi)$:
1. $t_\alpha\text{'} = [\ \text{AES}(kc + kv, \mathbf{0})\ , \text{AES}(kc + kv, \text{iv}\|\mathbf{1})\ ]$
2. $\mathbf{ct}\text{'} = \text{AES}(kc + kv, \mathbf{pt})$; $\text{c}_k\text{'} = \text{commit}(kc)$
3. assert: $t_\alpha \overset{?}{=} t_\alpha\text{'}, \text{c}_k \overset{?}{=} \text{c}_k\text{'}, \mathbf{ct} \overset{?}{=} \mathbf{ct}\text{'}, 1 \overset{?}{=} f_\phi(\mathbf{pt})$
```

Figure 5: Circuit logic of ZKPs in the client challenge. The semicolon ; separates private inputs (left side) from public inputs (right side). The function f_ϕ evaluates conditions expressed by a public statement ϕ on the plaintext pt.

and two keys $k_1 \neq k_2$ such that encrypting m_1 under k_1 and encrypting m_2 under k_2 yield the same ciphertext ct and tag t [13]. This attack is problematic and would break *session-integrity* and, with that, *session-authenticity*. In other words, a successful attack allows the *client* to prove arbitrary TLS data as TLS-authentic in the client challenge. TLS oracles solve this attack by letting the *client* disclose a commitment of the key share to the *verifier* (cf. Figure 4) [59]. The extra commitment binds the *client* to a fixed key share, which is verified during the client challenge. Fixing the key share must happen before the *verifier* discloses remaining session secrets (e.g. $\mathbf{kv_X}$ in Figure 4). Otherwise, an attacker can arbitrarily compute valid authentication tags and ciphertext chunks, which is a prerequisite to perform the attack [13].

4.1.3 *Client Challenge in Asymmetric Privacy Setting.*

Once the *client* has gathered enough TLS data, the *verifier* reveals remaining secret shares to the *client* (cf. Figure 4). When the *client* obtains full access to session secrets, an asymmetric privacy setting between the *client* and *verifier* is established. Now, the *client* is able to access TLS data by decrypting exchanged records which the *verifier* cannot.

To preserve *session-integrity*, the *verifier* confronts the client with irreversible challenges via ZKP circuits (cf. Figure 5). For cipher suites running MtE, the *client* must prove that the plaintext evaluates against the authentication tag which is encrypted under the last three ciphertext chunks. For AEAD cipher suites, the *client* shows that the secret key share kc (i) maps to the previously shared key share commitment, (ii) connects plaintext and ciphertext chunks, and (iii) evaluates to intermediate values t_α for the tag computation. Here, the *verifier* validates public ZKP inputs (e.g. t_α) out-of-circuit, which have been shared by the *client*.

Current TLS oracles rely on proof systems (e.g. *Groth16*), which efficiently evaluate algebraic or zkSNARK-friendly arithmetic [10, 18, 53, 57, 59]. However, the ZKP circuits of TLS oracles (cf. Figure 5) heavily depend on legacy algorithms (e.g. AES or SHA256) which rely on zkSNARK non-friendly, non-algebraic arithmetic.

Insight 2: Proof systems are not tailored to the arithmetic requirements and the privacy setting found in TLS oracles.

4.2 HVZK and Asymmetric Privacy

This section picks up on our second insight (cf. Section 4.1.3) and formalizes asymmetric privacy. In the asymmetric privacy setting, we secure a HVZK proof system, which efficiently proves non-algebraic statements, against malicious adversaries. Subsequently, we show how our formal definitions apply to TLS.

4.2.1 *Formalizing Asymmetric Privacy.*

In the scope of this work, we formalize asymmetric privacy in a setting with three parties; parties p_1 and p_2 and a trusted dealer d. We rely on a maliciously secure 2PC scheme Π_{2PC}, a secure commitment scheme Π_{Com}, and a secret sharing scheme Π_{SS} (cf. Appendix A for formal definitions).

To set up an asymmetric privacy setting between p_1 and p_2, the dealer d calls Π_{SS}.**Share** and individually shares r_1 with p_1 and r_2 with p_2. It holds that the secret shares r_1+r_2 sum to r. We define two cases to commit a message string m into a commitment string c using r. The first case requires p_1 and p_2 to execute a circuit C in the 2PC scheme Π_{2PC}, where C calls Π_{SS}.**Reconstruct** and Π_{Com}.**Commit**. In this case, p_1 inputs m and r_1 and p_2 inputs r_2. After the commitment c has been computed and disclosed, p_2 releases the secret share r_2 to p_1, and, with that, initiates the asymmetric privacy setting. Now, p_1 can reconstruct r. With access to m and r, only p_1 is capable of successfully proving Π_{Com}.**Open**.

For the second case, the trusted dealer computes and discloses the commitment string c on a message string m with randomness r. If the trusted dealer performs the commitment, then the dealer additionally shares the message string m with a party (e.g. with p_1). To set up the asymmetric privacy setting, p_2 discloses the secret share r_2 after receiving the commitment string c from the dealer. In the second case, the dealer and p_1 have access to r and can prove a successful commitment opening to p_2.

4.2.2 *HVZK and Selective-failure Attacks.*

To improve the performance of proof computations during the client challenge (cf. Figure 4), we deploy a HVZK proof challenge to evaluate the circuits of Figure 5. We consider the asymmetric privacy setting between p_1 as the *client* and p_2 as the *verifier*, where p_1 has access to all TLS session secrets. The proof system of the work [29] uses semi-honest 2PC based on boolean garbled circuits to achieve the notion of HVZK and assumes an honest *verifier* (cf. Section 2.3.4). However, in a setting with malicious adversaries, semi-honest 2PC is susceptible to selective failure attacks [52]. Notice that if a malicious p_2 intentionally corrupts one or multiple rows of the garbling tables, p_2 can learn information on which row has been evaluated by p_1. On top and with knowledge of the row permutations, p_2 is capable of deriving secret information of p_1's inputs. In the following subsection, we introduce a secure validation protocol which is unilaterally performed by p_1. The validation detects a maliciously acting p_2 before any secrecy leakage occurs.

4.2.3 *Unilateral Secure Validation.*

The unilateral secure validation is performed once p_1 obtains all public semi-honest 2PC parameters of the HVZK proof system [29], which comprise garbled tables $G(C_{\text{HVZK}})$ and external labels $\mathbf{e}$ (cf. Section 2.3.1). The parties p_2 and p_1 exchange wire keys $\mathbf{k}$ corresponding to the private inputs $\mathbf{pt}$ via the OT^1_2 oblivious transfer protocol [11] and p_2 omits sharing the output label decoding table $\boldsymbol{T_{l-d}}$. The party p_2, acting as the garbler and *verifier*, is convinced of the HVZK proof if p_1, acting

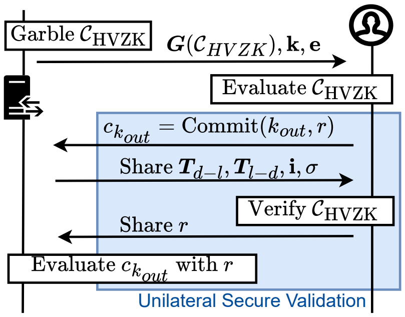

Garble C_{HVZK}
$\boldsymbol{G}(\mathcal{C}_{HVZK}), \mathbf{k}, \mathbf{e}$
Evaluate C_{HVZK}
$c_{k_{out}} = \text{Commit}(k_{out}, r)$
Share $\boldsymbol{T}_{d-l}, \boldsymbol{T}_{l-d}, \mathbf{i}, \sigma$
Verify C_{HVZK}
Share r
Evaluate $c_{k_{out}}$ with r
Unilateral Secure Validation

Figure 6: Secure unilateral validation protocol in the asymmetric privacy setting to assert correct garbling of C_{HVZK}.

as the evaluator and *client*, returns the output wire key that corresponds to the output bit 1. Depending on the cipher suite, the 2PC circuit C_{HVZK} implements the logic of the circuits C_{MtE} or C_{AEAD} (cf. Figure 5) and yields a 1 if all assertions are satisfied.

In the default HVZK proof system [29], the *client* p_1 must return the output wire key back to the *verifier* p_2 to complete the HVZK proof protocol. However, to achieve security in a malicious setting, we require p_1 to run a new secure validation phase (cf. Figure 6). The unilateral validation enforces p_1 to share a commitment $c_{k_{out}}$ of the output wire key. After sharing the commitment $c_{k_{out}}$, p_2 discloses all garbling parameters of the semi-honest 2PC computation with p_1. Revealing all garbling parameters allows p_1 to verify if C_{HVZK} has been garbled correctly by recomputing the garbled circuit. And, due to the asymmetric privacy setting, p_1 learns nothing new because all TLS session secrets of p_2 have already been shared with p_1. If p_1 detects a malicious garbling, then p_1 aborts the protocol. Otherwise, p_1 discloses the commitment randomness r such that p_2 can verify the correct output wire key via $c_{k_{out}}$. We show the security of this construction in the Appendix B.1.

4.2.4 TLS Compatibility. Our formalization is compatible with the typical TLS oracle setting with a single *verifier*. The *server* takes over the role of the trusted dealer to set up multiplicative secret shares between the client parties via the 3PHS. Subsequently, the ECTF protocol converts client secret shares into an additive representation. The *client* and *verifier* collaboratively commit to TLS session parameters by computing authentication tags and ciphertext chunks (cf. first case commit in Section 4.2.1). Otherwise, the client-side parties receive commitments by capturing server-side traffic (cf. second case commit in Section 4.2.1). Remember that if a cipher suite is not key-committing (e.g. AEAD cipher suites), then the *verifier* needs an additional key share commitment from the *client*. Access to secure commitments is a prerequisite for the asymmetric privacy setting. Next, the *verifier* initiates the asymmetric privacy setting by disclosing secret shares of TLS parameters to the *client*. From here on, only the *client* is capable of computing valid commitment openings. Thus, in the client challenge, a HVZK proof system with our unilateral validation protocol can be deployed.

5 Optimizing End-to-end Performance

Our second contribution applies to TLS oracles running the TLS 1.3 1-RTT mode and targets our first insight (cf. Section 4.1.2). In detail, we show how client parties can securely derive and authenticate the SHTS parameter in a malicious security setting (cf. Section 5.1). Subsequently, we leverage the SHTS authenticity to deploy an optimized garble-then-prove paradigm, which entirely relies on semi-honest 2PC techniques (cf. Section 5.2).

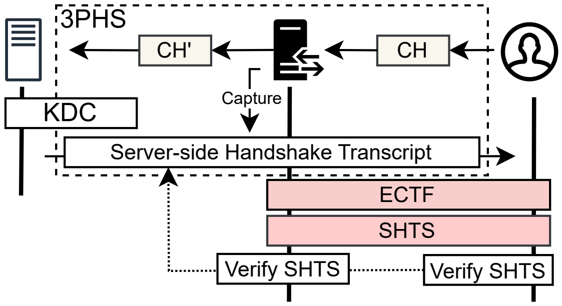


Figure 7: Mutual SHTS verification at client parties. Red boxes indicate values derived in maliciously secure 2PC protocols.

5.1 Authenticating SHTS

This section explains why pre-fetching cipher suites is a necessity to reliably validate SHTS authenticity. Further, we show how our SHTS validation sequence counters possible attacks.

5.1.1 Pre-fetch for Immediate Server-side Handshake Transcript. In the 1-RTT mode of TLS 1.3, *servers* immediately derive session secrets and return authenticated handshake messages upon the reception of compliant CH messages (cf. Figure 7). Even though TLS 1.3 allows the configuration of three AEAD cipher suites and two possible parameters for the key agreement (ECDHE with X25519 or P-256), *clients* may select an unsupported parameterization. In this case, TLS parameters must be renegotiated. To prevent any renegotiation, we expect *clients* to perform a single pre-fetch call to detect compatible configurations. This way, *clients* and *verifiers* can reliably launch a TLS oracle session and expect the *server* to return the handshake transcript after a single CH message.

The pre-fetch call is a fingerprinting technique which returns server-supported configurations of cipher suites. In this work, the pre-fetch call sends a CH message to the *server*. The requested TLS handshake session by the CH message is aborted once the server-side parameters are returned [32].

5.1.2 Compute and Disclose of SHTS. Once *clients* receive the server-side handshake traffic, both the *client* and *verifier* continue to derive secret-shared session parameters via the 3PHS (cf. Appendix A.1) and the ECTF protocol (cf. Appendix A.5.2) [43, 59]. In the end, the *verifier* locally maintains s_1 and the *client* locally keeps s_2 and it holds that $s_1 + s_2 = \text{DHE}$. To derive SHTS, the *verifier* and *client* evaluate the circuit C_{SHTS} (cf. Table 2) in a maliciously secure 2PC system. Similar to the works [18, 21, 59], we leverage the fact that, during the handshake phase, the *client* can securely disclose the SHTS parameter to the *verifier*. Even though the *verifier* knows SHTS, the key independence property of TLS 1.3 prevents the *verifier* from learning the HS secret [15], as HS is protected by

hkdf.exp (cf. line 5 and 17 of Figure 2). Without access to HS, the adversary cannot derive application traffic keys from HS.

5.1.3 Attacking SHTS Authenticity. The server-side handshake transcript contains the SF message, which can be seen as a commitment to established TLS session parameters [18, 21]. We require both client parties to capture the server-side handshake transcript before the *client* and *verifier* compute SHTS via the 2PC circuit C_{SHTS} (cf. Figure 7). This condition prevents adversaries from forging the authenticity of SHTS as client parties can validate handshake session secrets against the commitment (cf. Appendix B.2).

To provide more context, the following aspects must be considered. Our system model prevents the adversary (i) from compromising the *server*'s private key and (ii) from accessing full handshake secrets through the 3PHS and the ECTF protocol. Thus, to obtain a valid signature from the *server*, an adversary must replay a previous and individually established handshake transcript.

Malicious Client. If the adversary takes the role of the *client*, then the *verifier* injects fresh randomness by determining the CH transcript. As a consequence, a replayed handshake signature does not match the new transcript and the adversary cannot sign a new transcript without the *server*'s private key. Thus, in this scenario, the signature verification detects malicious behavior.

Malicious Verifier. If the adversary acts as a *verifier*, then the adversary determines the CH message transcript which the *client* cannot. This gives the adversary the opportunity to replay a previously established handshake session, where the adversary knows the session secret DHE. If we allow the computation of SHTS before capturing the server-side handshake transcript, the following attack is possible[2]: The adversary picks a random input to compute SHTS' and recomputes the last part of a previously established handshake transcript using SHTS' (cf. lines 8,9,11 of Figure 2). Once the adversary shares the forged server-side handshake transcript, the *client* accepts because the SF validation succeeds. Afterwards, the adversary accepts any incoming requests from the *client* which could contain confidential data (e.g. credentials). This attack compromises *session-authenticity* and *session-integrity*. However, if the *client* honestly proceeds the 2PC computations according to TLS 1.3, then *session-confidentiality* on record phase data holds.

We close this attack with the previously defined condition, where client parties capture the server-side handshake transcript before the joint evaluation of SHTS. Here, the adversary is faced with the following challenge. The adversary must replay a handshake transcript which complies with the prospective SHTS value. Since the adversary cannot predict the outcome of C_{SHTS} without access to the secret s_2 of the *client*, the adversary has negligible chances in guessing a compliant SF message beforehand (cf. Appendix B.2). This way, *session-authenticity* and *session-integrity* hold.

5.1.4 SHTS Validation. To validate SHTS, the *client* and *verifier* match the locally computed SF' values against the SF value from the *server*. To access SF, both client parties derive the handshake traffic secrets k_{SHTS}, iv$_{\text{SHTS}}$ and decrypt server-side handshake messages. Additionally, the client parties assert the validity of the *server*'s certificate. Afterwards, the client side jointly computes C_{CHTS} using maliciously secure 2PC, which outputs the client handshake traffic secret (CHTS) to the *client*. With CHTS, the *client* completes the handshake by computing and sharing the CF message.

Table 2: Mapping of TLS Computation traces to 2PC circuits. We use XHTS for SHTS/CHTS and XATS for SATS/CATS.

Circuit	Computation Trace
C_{XHTS}	DHE=s_1+s_2; DHE to XHTS
$C_{(\text{k,iv})}$	DHE=s_1+s_2; DHE to (kc_{XATS}, kv_{XATS}, iv$_{\text{XATS}}$)
$C_{\text{CB}_{2+}}$	(kc_{XATS}, kv_{XATS}, iv$_{\text{XATS}}$) to CB$_{2+}$
C_t	(kc_{XATS}, kv_{XATS}, iv$_{\text{XATS}}$, ct) to t
C_{open}	DHE=s_1+s_2; DHE to SHTS; DHE to CB

5.2 Garble-then-prove with Semi-honest 2PC

We apply a modified garble-then-prove paradigm [53]. In the garble phase, we replace 2PC computations based on authenticated garbling with lightweight semi-honest 2PC computations that do not require authenticated garbling. We show the security of our construction in the Appendix B.3.

5.2.1 Intuition of Garble-then-Prove. The idea behind the garble-then-prove paradigm is as follows. If a malicious *client* acts as the garbler of the semi-honest 2PC system, then the *client* can mount selective failure attacks throughout the record phase. However, as TLS oracles eventually disclose session secrets of the *verifier*, the malicious *client* learns nothing beyond what the honest *client* would have learned. The prove phase is supposed to (i) detect any cheating activities of the *client* and (ii) provide the *verifier* with a conditional abort option before any data attestations occur. To do so, the prove phase recomputes and compares all semi-honest 2PC computations of the *client* against securely authenticated session parameters at the *verifier* (cf. Section 5.2.3).

5.2.2 Garble Phase. In the garble phase, the *client* and *verifier* collaboratively evaluate multiple 2PC circuits (cf. Table 2), where the *client* acts as the garbler and the *verifier* acts as the evaluator. In the handshake phase, the circuit $C_{(\text{k,iv})}$ yields secret-shared application traffic secrets to the *client* and *verifier*. The additive relation of secrets (e.g. k_{XATS}=kc_{XATS}+kv_{XATS}) continues to hold.

In the record phase, the 2PC circuit $C_{\text{CB}_{2+}}$ outputs counter blocks CB$_i$ (cf. Figure 11) to the *client*, with i>1. To prevent commitment attacks on records, no block CB$_i$ ever includes any CB$_0$,CB$_1$ blocks. To encrypt a request, the *client* computes $\mathbf{ct}_{req}$ = $\mathbf{CB}_{2+} \oplus \mathbf{pt}_{req}$ and shares the ciphertext $\mathbf{ct}_{req}$ with the *verifier*. Next, the client parties jointly compute t_{req} using C_t and the *verifier* sends the request to the *server*. After client parties receive a response ($\mathbf{ct}_{resp}$, t_{resp}), the *verifier* discloses all session secrets and initiates the asymmetric privacy setting. Notice that computing the AEAD key commitment c_k is redundant because, in the client challenge, we can consider SHTS as an authenticated commitment on session secrets.

5.2.3 Prove Phase. The prove phase starts with the asymmetric privacy setting (cf. Section 4.2), where the *verifier* has captured the TLS 1.3 transcript and disclosed session secrets to the *client*. The prove phase of this work considers the authenticated SHTS

[2] This attack applies to the wok [59] in the TLS 1.3 mode. The work [10] indicates this attack. Our work provides the first full attack description and proposes a countermeasure.

```
$C_{\mathbf{zkOpen}}(s_2, \mathbf{pt}\ ;\ s_1, \mathbf{I}^{\text{open}}, \mathbf{ct}, t, \text{SHTS}, \phi)$:
1. $\text{SHTS}', \mathbf{CB}' = C_{\text{open}}(s_1, s_2, \mathbf{I}^{\text{open}})$
2. $\mathbf{ct}' = \mathbf{CB}' \oplus \mathbf{pt}, t' = [\ \text{CB}_0'\ , \text{CB}_1'\ ]$
3. assert: $\text{SHTS}' \overset{?}{=} \text{SHTS}, \mathbf{ct}' \overset{?}{=} \mathbf{ct}, t' \overset{?}{=} t, 1 \overset{?}{=} f_\phi(\mathbf{pt})$
$C_{\mathbf{tpOpen}}(s_2\ ;\ s_1, \mathbf{I}^{\text{open}}, \mathbf{CB}, t, \text{SHTS})$:
1. $\text{SHTS}', \mathbf{CB}' = C_{\text{open}}(s_1, s_2, \mathbf{I}^{\text{open}}), t' = [\text{CB}_0', \text{CB}_1']$
2. assert: $\text{SHTS}' \overset{?}{=} \text{SHTS}, t' \overset{?}{=} t, \mathbf{CB} \overset{?}{=} \mathbf{CB}'$
```

Figure 8: Extending the ZKP circuit C_{AEAD} (cf. Figure 5) to the HVZK circuit used by our E2E-optimized TLS 1.3 oracle for the privacy-preserving and transparent client challenges.

parameter as a commitment string. Further, the 2PC circuit C_{HVZK} of the client challenge is set to the $C_{\mathbf{zkOpen}}$ algorithm (cf. Figure 8). The circuit C_{open} derives SHTS and counter blocks $\mathbf{CB}_i$, where the list of indices $\mathbf{I}^{\text{open}}$ indicates the plaintext/ciphertext chunks of interest. The *client* determines $\mathbf{I}^{\text{open}}$ according to plaintext chunks that are passed to f_ϕ. Notice that the assertion of the authentication tag is reduced to comparing intermediate values CB_0, CB_1. The in-circuit derivation of CB_0, CB_1 hides application traffic keys from the *verifier* and frees $C_{\mathbf{zkOpen}}$ from expensive algebraic operations (e.g. multiplication of Galois field (GF) polynomials). Remember that the HVZK proof system which evaluates $C_{\mathbf{zkOpen}}$ runs the unilateral secure validation to detect a malicious *verifier* (cf. Section 4.2). After the client challenge, the *verifier* derives t_{resp}' from t' out-of-circuit and asserts if $t_{resp}' \overset{?}{=} t_{resp}$. If all assertions succeed, the *verifier* attests the TLS 1.3 data of the *client* (cf. Section 5.3.3).

5.3 Additional Considerations

The following aspects complete the context of TLS 1.3 oracles beyond our contributions in Section 5.1 and 5.2. We provide complete E2E oracles with *Janus* optimizations in the Appendix D.

5.3.1 Operation Modes. TLS oracles can be operated in two different modes, which introduce distinct arrangements in the prove phase. Both operation modes depend on a list of indices $\mathbf{I}^{\text{open}}$, which the *client* shares to the *verifier*. $\mathbf{I}^{\text{open}}$ contains indices of TLS record chunks that are of interest for the statement evaluation in f_ϕ. For example, if the *client* wants to prove TLS data of the chunk pt_3, then the index i=3 is included in the list $\mathbf{I}^{\text{open}}$. The *client* selectively determines $\mathbf{I}^{\text{open}}$ once session secrets are obtained and records are decrypted. The *verifier* uses $\mathbf{I}^{\text{open}}$ to identify public inputs in form of ciphertext chunks for the verification of C_{HVZK} (cf. Figure 8).

Transparent Mode In the *transparent* mode, the *verifier* checks (i) if the AEAD encryption of presented plaintext chunks matches the captured ciphertext transcript and (ii) if a computation trace from the encryption key to SHTS exists. To prevent the *verifier* from learning TLS encryption keys, the *transparent* mode requires an adapted circuit $C_{\mathbf{tpOpen}}$ (cf. Figure 8). $C_{\mathbf{tpOpen}}$ takes as public input counter blocks $\mathbf{CB}$, which have been computed and shared by the *client*. Further, $C_{\mathbf{tpOpen}}$ asserts SHTS, counter blocks $\mathbf{CB}$, and authentication tags via intermediate values. The assertions $1 \overset{?}{=} f_\phi(\mathbf{pt})$ and $\mathbf{ct} \overset{?}{=} \mathbf{ct}' = \mathbf{CB} \oplus \mathbf{pt}$ are computed out-of-circuit because plaintext chunks are publicly disclosed.

Privacy-preserving Mode In the *privacy-preserving* mode, the *client* does not share $\mathbf{pt}$. Instead, the *client* shares $\mathbf{I}^{\text{open}}$ and proves knowledge of authentic plaintext data via the HVZK proof system. To do so, the *client* evaluates the 2PC circuit $C_{\mathbf{zkOpen}}$ and applies the unilateral secure validation.

5.3.2 Processing Multiple Records. Concerning the collaborative processing of multiple records in the *record phase*, we differentiate computations with respect to the following dependencies:

Requests are independent of responses. If no request depends on the contents of a response, then the circuit $C_{\text{CB}_{2+}}$ is only called for the compilation of requests. Response counter blocks (CBs) can be locally computed by the *client* once the asymmetric privacy setting enforces the disclosure of full session secrets to the *client*.

Requests depend on responses. If a request of number $n > 1$ depends on the contents of responses $\mathbf{ct}=[ct_1, \ldots, ct_l]$, where each response ct_m has an index $m < n$, then the *client* and *verifier* perform l executions of the circuit $C_{\text{CB}_{2+}}$. The evaluation of l circuits $C_{\text{CB}_{2+}}$ yields l vectors of encrypted counter blocks $\mathbf{CB}_{2+}$ to the *client*. With l vectors of $\mathbf{CB}_{2+}$, the *client* is capable of accessing the contents of the responses $\mathbf{ct}=[ct_1, \ldots, ct_l]$ to construct the n-th request. To preserve *MITM-resistance* and prevent commitment attacks, it must hold that the *verifier* intercepts the pair $(\mathbf{ct}, t)$ before the circuit $C_{\text{CB}_{2+}}$ outputs the corresponding $\mathbf{CB}_{2+}$ of response $\mathbf{ct}$.

5.3.3 Data Attestation. The attestation structure varies according to the operation modes. In the transparent mode, the *verifier* hashes verified TLS data and signs the hash. Here, the certification parameter $p_{cert}=(t, \phi, pk, \sigma)$ includes a signature $\sigma=\mathbf{ds.Sign}(sk,[\phi, t])$ computed at time t. The *verifier* overwrites the statement $\phi = H(\mathbf{pt})$ to the hash of verified data such that every third party can evaluate presented TLS data against ϕ and against arbitrary statements. In the privacy-preserving mode, the structure of p_{cert} remains the same except that the statement ϕ expresses asserted constraints. This attestation ensures that the *verifier* successfully validated private TLS data against the statement ϕ at time t. The certificate p_{cert} enables verifiable provenance of TLS data because p_{cert} can be verified by any third party who trusts the *verifier* and the *server*.

6 Performance Evaluation

The evaluation describes the software stack and measures the impacts of our two contributions; The first contribution improves proof computation times for TLS 1.2/1.3 oracles. The second contribution improves the E2E performance of TLS 1.3 oracles. We provide micro benchmarks on a circuit level in the Appendix C.

6.1 Implementation

Tooling: We implement the 3PHS by modifying the Golang *crypto/tls* standard library[3] and configure the NIST P-256 elliptic curve for the elliptic curve Diffie–Hellman exchange (ECDHE). Our ECTF conversion algorithm uses the Golang Paillier cryptosystem[4]. We use the *mpc* library[5] to access semi-honest 2PC based on garbled circuits, which supports the optimizations free-XOR [31], fixed-key AES garbling (AES-NI instruction set) [5], and half-gates [56]. We

[3] https://pkg.go.dev/crypto/tls

[4] https://github.com/didiercrunch/paillier

[5] https://github.com/markkurossi/mpc

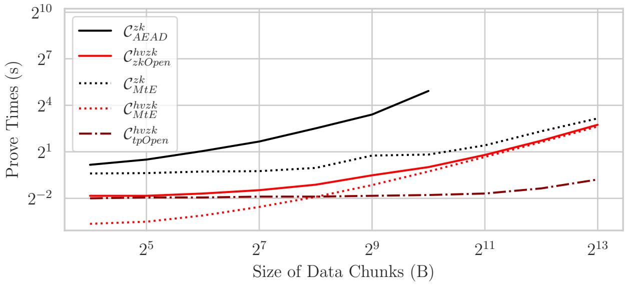


Figure 9: Scalability analysis of ZKP circuits, where circuits C_{MtE} (dotted) are compatible with TLS 1.2 only. Circuits C^{hvzk} leverage *Janus* optimizations. Lines closer to the bottom right corner are "better" and prove more data in less time.

adjust *mpc* to output single wire labels if we execute 2PC circuits in the context of the HVZK proof systems. We rely on the *ag2pc* framework[6] to implement maliciously secure 2PC circuits in TLS 1.2. To compute ZKPs, we rely on the *gnark* framework [8]. We open-source our secure computation circuits[7]. Based on our cipher suite analysis (cf. Appendix C.1), we implement the secure computation circuits AES128 and SHA256 to achieve compatibility with the most popular TLS 1.2/1.3 cipher suites.

E2E Benchmarks: The numbers of *TLSn*[8] and *DiStefano*[9] are reproduced by running publicly available experiments (cf. Table 4). Due to the fact that Deco [59] is closed source, we open source our $Deco_{zk}$ re-implementation[10], which executes TLS 1.2 configured with CBC-HMAC. The implementation of *Janus12* is equal to $Deco_{zk}$ except for the post-record phase. Here, *Janus12* employs the HVZK proof system implemented with the *mpc* framework. The TLS 1.3 oracles *DecoProxy* and *Janus13* rely on AEAD cipher suites which we implement with the *mpc* library. Malicious 2PC circuits computed with the *mpc* framework use the dual-execution mode [27]. We rely on *gnark* to implement ZKP circuits for *Deco*, *DecoProxy*, and *Origo*.

6.2 Performance

All performance benchmarks have been averaged over ten executions and have been collected on a MacBook Pro configured with the Apple M1 Pro chip and 32 GB of random access memory (RAM).

6.2.1 *Client Challenge Benchmarks.*

Concerning our first optimization (cf. Section 4), we evaluate ZKP circuits that are used during the client challenge. We execute the traditional circuits C^{zk}_{AEAD}, C^{zk}_{MtE} (cf. Figure 5) as a baseline using the fastest *gnark* proof system *Groth16*. We execute the circuits C^{hvzk}_{zkOpen} as an AEAD variant with the SHTS assertion (cf. Figure 8) and C^{hvzk}_{MtE} using the HVZK proof system. We depict the protocol support of the circuit variants in Table 3. Generally, the HVZK circuits (cf. red in Figure 9) achieve the best performance, where MtE-based circuits are ahead of AEAD circuits.

Table 3: Mapping protocols to cipher suite modes which support the ECDHE_ECDSA_AES128_SHA256 configuration.

Mode	Variant	Protocols
12_GCM	AEAD	*TLSn*
12_CBC	MtE	*Deco*,*Janus12*
13_GCM	AEAD	*DiStefano*, *DecoProxy*, *Origo*, *Janus13*

This makes sense as the circuit C^{hvzk}_{zkOpen} requires additional logic of constant size to derive and validate keys against SHTS. For larger data sizes, this overhead diminishes as AES (e.g. AEAD circuits) or SHA256 (e.g. MtE circuits) dominate the circuit complexity. Further, we benchmark the transparent validation of TLS data via C_{tpOpen} (cf. Figure 8), which outperforms the privacy-preserving validation for data sizes beyond 500 bytes.

6.2.2 *Optimized End-to-end Performance.*

We present E2E benchmarks of open-source TLS oracles in Table 4 and provide additional micro benchmarks in the Appendix C. Concerning TLS 1.2 (cf. top part of Table 4), we run *Janus12* using CBC-HMAC to benefit from constant size circuits in the record phase. For *TLSn*, which runs TLS 1.2 using AEAD, record phase 2PC complexity is determined by the size of the record (cf. row 1 vs row 2/3 in Table 4). Even though the post-record communication increases, *Janus12* achieves the fastest post-record execution benchmarks. For instance, the HVZK proof computations of *Janus12* outperform related works by a factor of 9 in the local area network (LAN) setting.

Concerning TLS 1.3, *Origo* and *DecoProxy* circumvent 2PC computation of the record phase by introducing an additional trust assumption (*clients* cannot mount MITM attacks). As a result, these works behave equal to the TLS 1.3 baseline in the record phase. *DiStefano* sets the fastest handshake execution times, which we link to the enhanced Multiplicative to Additive (MtA) algorithm in the ECTF protocol [10, 54]. Our implementation does not incorporate handshake optimizations proposed by *DiStefano* yet, because we achieve practical handshake benchmarks with our Paillier-based MtA conversion of session secrets. *Janus13* runs an AEAD cipher suite and evaluates 8 times more data (256b request, 2kb response) while remaining practical in all protocol phases.

7 Discussion

The discussion presents related works and summarizes remaining limitations and future work directions.

7.1 Related Works

The work *XYWY23* [53] introduce the garble-then-prove paradigm based on semi-honest 2PC with authenticated garbling. After the garble phase, authenticated garbling bits are transformed into a Pedersen commitment which can be opened in a zkSNARK proof system. By contrast, our E2E optimization for TLS 1.3 derives SHTS authenticity in a malicious setting. In the garble phase, we deploy semi-honest 2PC without authenticated garbling.

The work *Zombie* [57] notices that legacy algorithms constitute over 40% of TLS computations and decouples stream cipher computations with a *pad commitment.* The *pad commitment* is used to partly outsource legacy algorithms from the zkSNARK circuit

[6] https://github.com/emp-toolkit/emp-ag2pc
[7] https://github.com/jplaui/circuits_janus
[8] https://github.com/tlsnotary/tlsn/tree/main
[9] https://github.com/brave-experiments/DiStefano/tree/main
[10] https://github.com/jplaui/decoTls12MtE

Table 4: End-to-end (E2E) benchmarks of open-sourced TLS oracles. For the LAN setting, we assume a round-trip-time RTT=0 ms and a transmission rate r_t=1 Gbps. The wide area network (WAN) setting assumes a round-trip time (RTT) =50 ms and r_t=100 Mbps. Protocols starting with an * require an additional security assumption. Works marked with a ' use a transparent setup assumption to compute ZKPs. Entries marked with " take over the value of the entry above.

			Communication (kb)				Execution LAN (s) / WAN (s)			
Protocol	**Data**	**vTLS**	Offline	Handshake	Record	Post-record	Offline	Handshake	Record	Post-record
TLS	-	1.2	-	1.6	0.67	-	- / -	0.32 / 0.72	0.3 (ms) / 0.2	- / -
TLSn$_{\text{tp}}$	579b	1.2	178 (MB)	36 (MB)	40 (MB)	0.57	3.3 / 17.54	1.18 / 4.46	1.12 / 4.52	0.76 / 0.9
Deco$_{\text{zk}}$	32b	1.2	523.51 (MB)	294.52	150.55	0.23	14.38 / 56.26	0.94 / 2.36	0.68 / 1.59	0.76 / 0.86
'*Janus12*$_{\text{zk}}$	32b	1.2	415.22 (MB)	"	"	28.13	2.9 / 36.1	" / "	" / "	0.08 / 0.23
TLS	-	1.3	-	1.42	0.71	-	- / -	0.36 / 0.76	0.49 (ms) / 0.2	- / -
**Origo*$_{\text{zk}}$	32b	1.3	367.61 (MB)	"	"	0.24	29.16 / 58.56	" / "	" / "	1.26 / 1.36
**DecoProxy*$_{\text{zk}}$	32b	1.3	578.47 (MB)	307.7	"	0.27	24.34 / 70.61	0.95 / 2.37	" / "	1.35 / 1.45
DiStefano	256b	1.3	220.484 (MB)	343.42	48.82	-	5.85 / 23.48	0.43 / 0.85	0.12 / 0.32	- / -
'*Janus13*$_{\text{tp}}$	2.2kb	1.3	305.17 (MB)	113.8	984	583	1.99 / 26.4	0.51 / 0.91	1.04 / 1.51	0.46 / 0.6
'*Janus13*$_{\text{zk}}$	2.2kb	1.3	406.29 (MB)	"	"	2 (MB)	2.63 / 35.13	" / "	" / "	2.08 / 2.34

Table 5: Related works feature comparison.

Paper	Type	Key Feature
TLSn/Deco	Notary	3PHS & 2PC-based TLS Client
Zombie	Proxy	Pad Commit
Origo	Proxy	2PC-free
DiStefano	Notary	Secure MtA & Browsing Privacy
XYWY23	Notary	Garble-then-prove & Pedersen Commit
Janus	Proxy	Secure 2PC-based hvZKP

to a pre-processing phase. We tackle the arithmetic requirements of legacy algorithms with a well-suited, 2PC-based HVZK proof system which we secure using an unilateral validation phase.

Another way to improve the efficiency of the *client* challenge is to decouple the maliciously secure 2PC evaluation of CBs, which is done in the works *DiStefano*, *TLSn*, *Deco* [10, 43, 59]. Notice that this optimization applies to TLS oracles which run AEAD cipher suites. The work *DiStefano* [10] secures a leakage of the Paillier-based MtA conversion protocol [51] and achieves browsing privacy, where the *verifier* does not lean which *servers* the *client* queries. The work *Origo* [18] proposes a 2PC-free oracle solution and reduces 2PC bandwidth overheads at the cost of requiring an additional security assumption (*clients* cannot mount MITM attacks). We summarize our feature analysis of related works in the Table 5.

7.2 Asymmetric Privacy & Related Concepts

Our definition of asymmetric privacy relates to the concept of a trapdoor hash function (THF) between two parties [14]. THFs guarantee function privacy for the sender and input privacy for the receiver. The private function evaluates receiver data at a private index. In contrast, our asymmetric privacy setting ensures input privacy for a sender and convinces the receiver of a public function which holds on the entire sender input. Further, to differentiate against other notions such as asymmetric differential privacy [49], our notion of asymmetric privacy targets a threshold number of parties with access to commitment secrets.

7.3 Limitations

A reliable execution of *Janus* optimizations for TLS 1.3 oracles requires an additional pre-fetch call. We recommend that operators of the *verifier* pre-fetch *servers* before launching oracle sessions. This way, *verifiers* can provide seamless attestation services for *clients*. We like to highlight that the *verifier* as a proxy is susceptible to blocking if server-side endpoints detect reoccurring Transmission Control Protocol (TCP) sessions from the same proxy IP [6, 57]. In this case, we recommend running the *verifier* as a notary service behind the *client* (as described by *TLSn*, *Deco* [43, 59]). Our work does not investigate compression techniques to lower 2PC bandwidth overheads. Neither does our work currently support the MtA optimizations pointed out by *DiStefano* [10] (1. ring-based MtA instead of Paillier-based MtA, 2. parallelized MtA for TLS 1.3 authentication tags). We consider these topics for future improvements.

7.4 Disclaimer: Legal and Compliance Issues

This section informs users and companies running the *Janus* TLS oracle about subsidiary conditions and agreements. As TLS oracles are *legacy-compatible*, companies running the *verifier* connect seamlessly to web endpoints which are queried by users. Web endpoints do not necessarily notice the *verifier*. Legal issues (e.g. copyright infringements) arise if users export proprietary content or declare false data ownership. In this case, companies are supposed to deny content. If companies operate oracles in the privacy-preserving mode, then companies learn nothing from transport data beyond the statement validity. In the transparent mode, companies can surveil opened data in plain. Users must be aware that companies learn network layer data (e.g. IP addresses, domains), which is required to operate the *proxy* service. Tracking or profiling oracle data may cause regulatory compliance violations.

7.5 Applications

Generally, the *Janus* optimizations make ZKP-computing *clients* practical in constrained environments (e.g. browsers, mobile, IoT). And, with that, serve existing oracle applications such as confidential financial instruments, legacy credentials, or the combating of price discrimination [59]. On top, our scalability benefits open new application fields where larger data sets or documents require proofs of provenance. In this context, our contributions help in fighting the dissemination of disinformation by attesting generative AI content, which is among the goals of the Coalition for Content Provenance and Authenticity (C2PA) [30, 46].

8 Conclusion

We reconsider the selection of secure computation techniques in TLS oracles by putting an emphasis on the asymmetric privacy setting and the conditions found in TLS 1.3. Concerning the asymmetric privacy setting, we show that a HVZK proof system can be deployed if the *client* performs a unilateral validation of the *verifier*. Concerning TLS 1.3 in the 1-RTT mode, we show that the authenticity of SHTS can lower algorithmic security requirements in the garble-then-prove paradigm. Our contributions improve the efficiency of ZKP computations and improve E2E benchmarks.

Acknowledgments

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002. This work has received funding from The Bavarian State Ministry for the Economy, Media, Energy and Technology, within the R&D program "Information and Communication Technology", managed by VDI/VDE Innovation + Technik GmbH.

References

[1] Carlisle Adams, Stephen Farrell, Tomi Kause, and Tero Mononen. 2005. *Internet X. 509 public key infrastructure certificate management protocol (CMP).* Technical Report.

[2] Nadhem J Al Fardan and Kenneth G Paterson. 2013. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE symposium on security and privacy.* IEEE, 526–540.

[3] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, International Association for Cryptologic Research, 191–219.

[4] Kar Balan, Shruti Agarwal, Simon Jenni, Andy Parsons, Andrew Gilbert, and John Collomosse. 2023. EKILA: Synthetic Media Provenance and Attribution for Generative Art. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 913–922.

[5] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy.* IEEE, 478–492.

[6] Karthikeyan Bhargavan, Ioana Boureanu, Antoine Delignat-Lavaud, Pierre-Alain Fouque, and Cristina Onete. 2018. A formal treatment of accountable proxying over TLS. In *2018 IEEE Symposium on Security and Privacy (SP).* IEEE, 799–816.

[7] Alexander R Block, Albert Garreta, Jonathan Katz, Justin Thaler, Pratyush Ranjan Tiwari, and Michal Zajac. 2023. Fiat-Shamir Security of FRI and Related SNARKs. *Cryptology ePrint Archive* (2023).

[8] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. 2023. *ConsenSys/gnark: v0.8.0.* https://doi.org/10.5281/zenodo.5819104

[9] Srdjan Capkun, Ercan Ozturk, Gene Tsudik, and Karl Wüst. 2021. Rosen: Robust and selective non-repudiation (for tls). In *Proceedings of the 2021 on Cloud Computing Security Workshop.* 97–109.

[10] Sofía Celi, Alex Davidson, Hamed Haddadi, Gonçalo Pestana, and Joe Rowell. 2023. DiStefano: Decentralized Infrastructure for Sharing Trusted Encrypted Facts and Nothing More. *Cryptology ePrint Archive* (2023).

[11] Tung Chou and Claudio Orlandi. 2015. The simplest protocol for oblivious transfer. In *Progress in Cryptology–LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings 4.* Springer, 40–58.

[12] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2015. Automated synthesis of optimized circuits for secure computation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* 1504–1517.

[13] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. 2018. Fast message franking: From invisible salamanders to encryptment. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38.* Springer, 155–186.

[14] Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. 2019. Trapdoor hash functions and their applications. In *Annual International Cryptology Conference.* Springer, 3–32.

[15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. 2015. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security.* 1197–1210.

[16] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. 2021. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology* 34, 4 (2021), 1–69.

[17] Jens Ernstberger, Jan Lauinger, Fatima Elsheimy, Liyi Zhou, Sebastian Steinhorst, Ran Canetti, Andrew Miller, Arthur Gervais, and Dawn Song. 2023. SoK: Data Sovereignty. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P).* IEEE, 122–143.

[18] Jens Ernstberger, Jan Lauinger, Yinnan Wu, Arthur Gervais, and Sebastian Steinhorst. 2024. Origo: Proving provenance of sensitive data with constant communication. *Cryptology ePrint Archive* (2024).

[19] Rosario Gennaro and Steven Goldfeder. 2018. Fast multiparty threshold ECDSA with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 1179–1194.

[20] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. 2021. Zero-Knowledge Middleboxes. *Cryptology ePrint Archive* (2021).

[21] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. 2022. {Zero-Knowledge} Middleboxes. In *31st USENIX Security Symposium (USENIX Security 22).* 4255–4272.

[22] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. 2017. Message franking via committing authenticated encryption. In *Annual International Cryptology Conference.* Springer, 66–97.

[23] Shay Gueron. 2020. Key Committing AEADs. *Cryptology ePrint Archive* (2020).

[24] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. Sok: General purpose compilers for secure multi-party computation. In *2019 IEEE symposium on security and privacy (SP).* IEEE, 1220–1237.

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

[26] Yan Huang. 2012. Practical secure two-party computation. *dated: Aug* (2012).

[27] Yan Huang, Jonathan Katz, and David Evans. 2012. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *2012 IEEE Symposium on Security and Privacy.* IEEE, 272–284.

[28] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security.* 85–96.

[29] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* 955–966.

[30] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. 2022. ZK-IMG: Attested Images via Zero-Knowledge Proofs to Fight Disinformation. *arXiv preprint arXiv:2211.04775* (2022).

[31] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35.* Springer, 486–498.

[32] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. 2018. Coming of age: A longitudinal study of tls deployment. In *Proceedings of the Internet Measurement Conference 2018.* 415–428.

[33] Julia Len, Paul Grubbs, and Thomas Ristenpart. 2021. Partitioning oracle attacks. In *30th USENIX security symposium (USENIX Security 21).* 195–212.

[34] Yehida Lindell. 2005. Secure multiparty computation for privacy preserving data mining. In *Encyclopedia of Data Warehousing and Mining.* IGI global, 1005–1009.

[35] Yehuda Lindell, Arash Afshar, Yi-Hsiu Chen, Iftach Haitner, and Samuel Ranellucci. 2022. Building MPC Wallets – Challenges and Solutions. https://cyber.biu.ac.il/wp-content/uploads/2022/11/Building-MPC-Wallets.pdf.

[36] Shayne Longpre, Robert Mahari, Anthony Chen, Naana Obeng-Marnu, Damien Sileo, William Brannon, Niklas Muennighoff, Nathan Khazam, Jad Kabbara, Kartik Perisetla, et al. 2023. The Data Provenance Initiative: A Large Scale Audit of Dataset Licensing & Attribution in AI. *arXiv preprint arXiv:2310.16787* (2023).

[37] Zhongtang Luo, Yanxue Jia, Yaobin Shen, and Aniket Kate. 2024. Proxying is Enough: Security of Proxying in TLS Oracles and AEAD Context Unforgeability. *Cryptology ePrint Archive* (2024).

[38] Dahlia Malkhi. 2023. *Exploring Proof of Solvency and Liability Verification Systems.* Retrieved August 28, 2023 from https://blog.chain.link/proof-of-solvency/

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

[40] Sanketh Menda, Julia Len, Paul Grubbs, and Thomas Ristenpart. 2023. Context Discovery and Commitment Attacks: How to Break CCM, EAX, SIV, and More. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 379–407.

[41] Anca Nitulescu. 2020. zk-SNARKs: a gentle introduction.

[42] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18*. Springer, 223–238.

[43] Privacy and Scaling Explorations. 2014-2024. TLSnotary: Proof of data authenticity. https://tlsnotary.org/.

[44] Christian Reitwiessner. 2016. zkSNARKs in a nutshell. *Ethereum blog* 6 (2016), 1–15.

[45] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, and Srdjan Capkun. 2017. TLS-N: Non-repudiation over TLS enabling-ubiquitous content signing for disintermediation. *Cryptology ePrint Archive* (2017).

[46] Leonard Rosenthol. 2022. C2PA: the world's first industry standard for content provenance. In *Applications of Digital Image Processing XLV*, Vol. 12226. SPIE, 122260P.

[47] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[48] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. 2015. *Recommendations for secure use of transport layer security (tls) and datagram transport layer security (dtls)*. Technical Report.

[49] Shun Takagi, Fumiyuki Kato, Yang Cao, and Masatoshi Yoshikawa. 2022. Asymmetric differential privacy. In *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, 1576–1581.

[50] Justin Thaler et al. 2022. Proofs, arguments, and zero-knowledge. *Foundations and Trends® in Privacy and Security* 4, 2–4 (2022), 117–660.

[51] Dmytro Tymokhanov and Omer Shlomovits. 2021. Alpha-rays: Key extraction attacks on threshold ecdsa implementations. *Cryptology ePrint Archive* (2021).

[52] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 21–37.

[53] Xiang Xie, Kang Yang, Xiao Wang, and Yu Yu. 2023. Lightweight Authentication of Web Data via Garble-Then-Prove. *Cryptology ePrint Archive* (2023).

[54] Haiyang Xue, Man Ho Au, Mengling Liu, Kwan Yin Chan, Handong Cui, Xiang Xie, Tsz Hon Yuen, and Chengru Zhang. 2023. Efficient Multiplicative-to-Additive Function from Joye-Libert Cryptosystem and Its Application to Threshold ECDSA. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2974–2988.

[55] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th annual symposium on foundations of computer science (Sfcs 1986)*. IEEE, 162–167.

[56] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II 34*. Springer, 220–250.

[57] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. 2023. Zombie: Middleboxes that Don't Snoop. *Cryptology ePrint Archive* (2023).

[58] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*. 270–282.

[59] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. 2020. Deco: Liberating web data using decentralized oracles for tls. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1919–1938.

A Cryptographic Building Blocks

We describe algorithmic constructions by introducing security properties and provide concise tuples of algorithms to explain input to output parameter mappings. For cryptographic protocols, we describe the inputs and outputs which are provided and obtained by involved parties. Additionally, we mention the security properties of exchanged parameters.

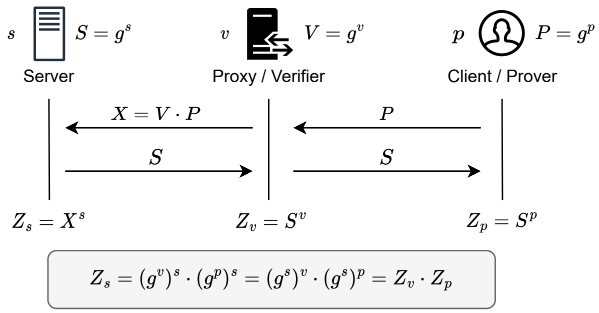


Figure 10: Illustration of the 3PHS and exchanged cryptographic parameters between the server, the proxy, and the client. The gray box at the bottom indicates the relationship between shared client-side secrets Z_v and Z_p, which corresponds to the session secret Z_s of the server.

A.1 Three-party Handshake

In the 3PHS (cf. Figure 10), each party picks a secret randomness (s, v, p) and computes its encrypted representation (S, V, P). By sharing $V + P = X$ with the server in the CH, the server derives the session secret $Z_s = s \cdot X$, which corresponds to the TLS 1.3 secret DHE. When the server shares S in the SH, both the proxy and client derive their shared session secrets Z_v and Z_p respectively such that $Z_s = Z_v + Z_p$ holds. In the end, neither the client nor the verifier have full access to the DHE secret of the TLS handshake phase. The 3PHS works for both TLS versions but in Figure 10, we show a TLS 1.3-specific configuration based on the ECDHE, where the parameters (e.g. Z_p) are EC points structured as $P = (x, y)$.

A.2 Digital Signatures

A digital signature scheme is defined by the following tuple of algorithms, where

- **ds.Setup**$(1^\lambda) \rightarrow (sk, pk)$ takes in a security parameter λ and outputs a public key cryptography key pair (sk, pk).
- **ds.Sign**$(sk, m) \rightarrow (\sigma)$ takes in a secret key sk and message m and outputs a signature σ.
- **ds.Verify**$(pk, m, \sigma) \rightarrow \{0, 1\}$ takes in the public key pk, a message m, and a signature σ. The algorithm outputs a 1 or 0 if or if not the signature verification succeeds.

By generating a signature σ on a fixed size message m with secret key sk, any party with access to the public key pk is able to verify message authenticity. Digital signatures guarantee that only the party in control of the secret key is capable of generating a valid signature on a message.

A.3 Keyed-hash or Hash-based Key Derivation Function

A HKDF function converts parameters with insufficient randomness into suitable keying material for encryption or authentication algorithms. The HKDF scheme is defined by a tuple of algorithms, where

- **hkdf.ext**$(s_{\text{salt}}, k_{\text{ikm}}) \rightarrow (k_{\text{pr}})$ takes in a string s_{salt}, input key material k_{ikm}, and returns a pseudorandom key k_{pr}.

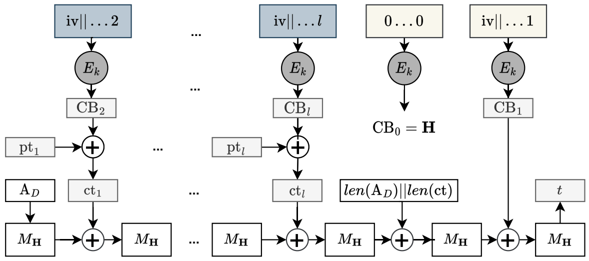


Figure 11: TLS 1.3 AEAD stream cipher in the GCM mode which encrypts a plaintext pt = $[pt_1, \dots, pt_l]$ to a ciphertext ct = $[ct_1, \dots, ct_l]$ under key k and authenticates the ciphertext ct and associated data A_D with the tag t.

- **hkdf.exp**(k_{pr}, s_{info}, l)⟶ (k_{okm}) takes in a pseudorandom key k_{pr}, a string s_{info} and a length parameter l and returns output key material k_{okm} of length l.

Both functions **hkdf.ext** and **hkdf.exp** internally use the **hmac** algorithm (cf. Formula 1), which takes in a key k, a bit string m, and generates a string which is indistinguishable from uniform random strings. The **hmac** algorithm requires a hash function H with input size b (e.g. b=64 if H=SHA256).

$$\begin{aligned} \mathbf{hmac}(k, m) = & H((k' \oplus opad)||H((k' \oplus ipad)||m)) \\ & \text{with } k' = H(k), \text{ if } len(k) > b \\ & \text{and } k' = k, \text{ else} \end{aligned} \tag{1}$$

A.4 Authenticated Encryption

AEAD provides communication channels with *confidentiality* and *integrity*. This means, exchanged communication records can only be read by parties with the encryption key and modifications of encrypted data can be detected. An AEAD encryption scheme is defined by the following tuple of algorithms, where

- **aead.Setup**(1^λ)⟶ (pp_{aead}) takes in the security parameter λ and outputs public parameters pp_{aead} of a stream cipher scheme E and authentication scheme A.
- **aead.Seal**(pp_{aead}, pt, k, a_D)⟶ (ct, t) takes in pp_{aead}, a plaintext pt, a key k, and additional data a_D. The output is a ciphertext-tag pair (ct, t), where $ct = E(pt)$ and $t = A(pt, k, a_D, ct)$ authenticates ct.
- **aead.Open**(pp_{aead}, ct, t, k, a_D)⟶ {pt, ∅} takes in pp_{aead}, a ciphertext ct, a tag t, a key k, and additional data a_D. The algorithm returns the plaintext pt upon successful decryption and validation of the ciphertext-tag pair, otherwise it returns an empty set ∅.

Stream ciphers are characterized by pseudorandom generators (AES in the GCM mode), which incrementally output key streams or CBs (cf. Figure 11). CBs are combined with plaintext data chunks to compute ciphertext data chunks. Subsequently, AEAD ciphers compute an authenticated tag t on all ciphertext chunks and associated data. The symbol $M_\mathbf{H}$ is a GF multiplication which translates bit strings into GF(2^{128}) polynomials, multiplies the polynomials modulo the field size, and translates the polynomial back to the bit string representation.

A.5 Secure Two-party Computation

Secure 2PC allows two mutually distrusting parties with private inputs x_1, x_2 to jointly compute a public function $f(x_1, x_2)$ without learning the private input of the counterparty. With that, secure 2PC counts as a special case of multi-party computation (MPC), with $m = 2$ parties and the adversary corrupting $t = 1$ parties [34]. The adversarial behavior model in 2PC protocols divides adversaries into semi-honest and malicious adversaries. Semi-honest adversaries honestly follow the protocol specification, whereas malicious adversaries arbitrarily deviate. In the following, we introduce secure 2PC protocols which are used in this work, and briefly introduce cryptographic constructions which are used to instantiate the secure 2PC protocols.

A.5.1 MtA Conversion based on Homomorphic Encryption. The secure 2PC MtA protocol converts multiplicative shares x, y into additive shares α, β such that $\alpha + \beta = x \cdot y = r$ yield the same result r. The MtA protocol exists in a vector form, which maps two vectors $\mathbf{x}, \mathbf{y}$, with a product $r = \mathbf{x} \cdot \mathbf{y}$, to two scalar values α, β, where the sum $r = \alpha + \beta$ is equal to the product r. The functionality of the vector MtA scheme can be instantiated based on Paillier additive Homomorphic Encryption (HE) [42]. Additive HE allows parties to locally compute additions and scalar multiplications on encrypted values. With the functionality provided by the Paillier cryptosystem, we define the vector MtA protocol, as specified in the work [19], with the following tuple of algorithms, where

- **mta.Setup**(1^λ)⟶ (sk_P,pk_P) takes in the security parameter λ and outputs a Paillier key pair (sk_P,pk_P).
- **mta.Enc**(**x**,sk_P)⟶ (**c1**) takes in a vector of field elements **x**=$[x_1, \dots, x_l]$ and a private key sk_P and outputs a vector of ciphertexts **c1**=$[E_{sk_P}(x_1), \dots, E_{sk_P}(x_l)]$.
- **mta.Eval**(**c1**,**y**,pk_P)⟶ (c_2,β) takes in the vector of ciphertexts **c1**=$[\text{c1}_1, \dots, \text{c1}_l]$, a vector of field elements **y**=$[y_1, \dots, y_l]$, and a public key pk_P. The output is a tuple of a ciphertext $\text{c2} = \text{c1}_1^{y_1} \cdot \dots \cdot \text{c1}_l^{y_l} \cdot E_{pk_P}(\beta')$ and the share $\beta = -\beta'$, where $\beta' \xleftarrow{\$} \mathbb{Z}_p$.
- **mta.Dec**(c2,sk_P)⟶ (α) takes as input a ciphertext c2 and a private key sk_P and outputs the share α=D_{sk_P}(c2).

The tuple of algorithms is supposed to be executed in the order where party p_1 first calls **mta.Setup** and **mta.Enc**. The function $E_k(z)$ is a Paillier encryption of message z under key k. After p_1 shares the public key pk_P and the vector of ciphertexts **c1** with party p_2, then p_2 calls **mta.Eval** and shares the ciphertext c2 with p_1. Last, p_1 calls **mta.Dec**, where $D_k(z)$ is a Paillier decryption of message z under key k. If the algorithms are executed in the described order, then party p_1 inputs private multiplicative shares in the vector **x** and obtains the additive share α. Party p_2 inputs the private vector of multiplicative shares **y** and obtains the additive share β. In the end, the relation $\mathbf{x} \cdot \mathbf{y} = \alpha + \beta$ holds, and neither the party p_1 nor the party p_2 learn anything about the private inputs of the counterparty.

A.5.2 ECTF Conversion. The ECTF algorithm is a secure 2PC protocol and converts multiplicative shares of two EC x-coordinates into additive shares [43, 59]. Figure 12 shows the computation sequence of the ECTF protocol which makes use the vector MtA

```
ECTF between two parties $p_1$ and $p_2$.
inputs: $P_1 = (x_1, y_1)$ by $p_1$, $P_2 = (x_2, y_2)$ by $p_2$.
outputs: $s_1$ to $p_1$, $s_2$ to $p_2$.
$p_1$: $(sk,pk)$=mta.Setup($1^\lambda$); send $pk$ to $p_2$
$p_1$: $\rho_1 \overset{\$}{\leftarrow} \mathbb{Z}_p$; c1=mta.Enc($[-x_1,\rho_1]$, $sk$);
    send c1 to $p_2$
$p_2$: $\rho_2 \overset{\$}{\leftarrow} \mathbb{Z}_p$;(c2,$\beta$)=mta.Eval(c1,$[\rho_2,x_2]$,$pk$);
    $\delta_2 = x_2 \cdot \rho_2 + \beta$; send (c2,$\delta_2$) to $p_1$
$p_1$: $\alpha$=mta.Dec(c2,$sk$);$\delta_1 = -x_1 \cdot \rho_1 + \alpha$;$\delta = \delta_1 + \delta_2$;
    $\eta_1 = \rho_1 \cdot \delta^{-1}$; c1=mta.Enc($[-y_1,\eta_1]$,$sk$);
    send (c1,$\delta_1$) to $p_2$
$p_2$: $\delta = \delta_1 + \delta_2$; $\eta_2 = \rho_2 \cdot \delta^{-1}$;
    (c2, $\beta$)=mta.Eval(c1,$[\eta_2,y_2]$,$pk$); $\lambda_2 = y_2 \cdot \eta_2 + \beta$;
    send c2 to $p_1$
$p_1$: $\alpha$=mta.Dec(c2,$sk$); $\lambda_1 = -y_1 \cdot \eta_1 + \alpha$;
    c1=mta.Enc($[\lambda_1]$,$sk$); send c1 to $p_2$
$p_2$: (c2,$\beta$)=mta.Eval(c1, $[\lambda_2]$, $pk$); $s_2 = 2 \cdot \beta + \lambda_2^2 - x_2$;
    send c2 to $p_1$
$p_1$: $\alpha$=mta.Dec(c2,$sk$); $s_1 = 2 \cdot \alpha + \lambda_1^2 - x_1$
```

Figure 12: The ECTF algorithm converts multiplicative shares in form of EC point x-coordinates from points $P_1, P_2 \in EC(\mathbb{F}_p)$ to additive shares $s_1, s_2 \in \mathbb{F}_p$. It holds that $s_1 + s_2 = x$, where x is the coordinate of the EC point $P_1 + P_2$.

algorithm defined in Section A.5.1. By running the ECTF protocol, two parties p_1 and p_2, with EC points P1, P2 as respective private inputs, mutually obtain additive shares s_1 and s_2, which sum to the x-coordinate of the EC points sum P1+P2. TLS oracles use the ECTF protocol to transform the client-side EC secret shares Z_v and Z_p into additive shares s_v and s_p [43, 59]. Since the relation $s_v + s_p = x$ for $(x, y) = Z_s$ holds, it becomes possible to follow the TLS specification by using secure 2PC based on boolean garbled circuits with bitwise additive shares as input.

A.5.3 Oblivious Transfer. Secure 2PC based on boolean GCs depends on the 1-out-of-2 OT_2^1 sub protocol to secretly exchange input parameters of the circuit [11]. The OT_2^1 involves two parties where party p_1 sends two messages m_1, m_2 to party p_2 and does not learn which of the two messages m_b is revealed to party p_2. Party p_2 inputs a secret bit b which decides the selection of the message m_b. An OT scheme is defined by a tuple of algorithms, where

- **ot.Setup**(1^λ)⟶ (pp_{OT}) takes as input a security parameter λ and outputs public parameters pp_{OT} of a hash function H and encryption schemes, where E_1/D_1 encrypts/decrypts based on modular exponentiation and E_2/D_2 encrypts/decrypts with a block cipher.
- **ot.TransferX**(pp_{OT})⟶ (X) takes in pp_{OT}, samples $x \overset{\$}{\leftarrow} \mathbb{Z}_p$, and outputs an encrypted secret $X = E_1(x)$.
- **ot.TransferY**(pp_{OT}, X, b)⟶ (Y, k_D) takes in pp_{OT}, a cipher X, a bit b, and samples $y \overset{\$}{\leftarrow} \mathbb{Z}_p$. The output is a decryption key $k_D = X^y$ and a cipher Y encrypting as $Y = E_1(y)$ if $b \overset{?}{=} 0$, or as $Y = X \cdot E_1(y)$ if $b \overset{?}{=} 1$.
- **ot.Encrypt**(pp_{OT}, X, Y, m_1, m_2, x)⟶ (**Z**) takes in pp_{OT}, Y, and derives $k_1 = H(Y^x)$, $k_2 = H((\frac{Y}{X})^x)$. The output is a vector of ciphers $\mathbf{Z} = [E_2(m_1, k_1), E_2(m_2, k_2)]$.
- **ot.Decrypt**(pp_{OT}, **Z**, k_D, b)⟶ (m_b) takes in pp_{OT}, key k_D, the bit b, and a vector of ciphers $\mathbf{Z} = [Z_1, Z_2]$. The output is the message $m_b = D_2(Z_b, k_D)$.

In the OT_2^1 protocol, party p_1 calls **ot.Setup** and **ot.TransferX**, and sends the public parameters and cipher X to p_2. Party p_2 calls **ot.TransferY**, locally keeps the decryption key and shares the cipher Y with p_1. Now, p_1 shares the output of **ot.Encrypt** with p_2, who obtains m_b by calling **ot.Decrypt**.

A.5.4 Semi-honest 2PC with Garbled Circuits. We define secure 2PC based on boolean garbled circuits by extending our OT definition of Section A.5.3 with the tuple of algorithms, where

- **gc.Setup**(1^λ)⟶ (pp_{GC}) takes in the security parameter λ and outputs public parameters pp_{GC}.
- **gc.Garble**(pp_{GC}, C_G, d_{in})⟶ ($\mathbf{k}_{in}^g$, **e**, $G(C)$, $T_{\text{k-d}}$, $T_{\text{d-k}}$) takes as input pp_{GC}, a boolean circuit C_G, the input bit string d_{in}, and randomly samples signal bits and wire keys σ,$k \overset{\$}{\leftarrow} \mathbb{Z}_n$. Every wire receives two wire keys where the internal labels map wire keys to the numbers 0 and 1. Based on the signal bits and internal labels, every wire receives two external labels. The output consists of input wire keys $\mathbf{k}_{in}$, the garbled tables $G(C)$, input and output decoding tables $T_{\text{d-k}}$, $T_{\text{k-d}}$, and external labels **e**.
- **gc.Evaluate**(pp_{GC}, $\mathbf{k}_{in}^g$, $\mathbf{k}_{in}^e$, **e**, $G(C)$)⟶ (k_{out}) takes in public parameters, input wire keys, external labels, and the garbled circuit tables and outputs output wire keys.

On a high-level, a 2PC system based on boolean garbled circuits involve a party p_1 as the garbler and party p_2 as the evaluator. Party p_1 calls **gc.Setup** and **gc.Garble**. Subsequently, p_1 sends **e**, $\mathbf{k}_{in}^g$, $G(C)$, and $T_{\text{k-d}}$ to p_2. If the semi-honest 2PC system is used in the context of an HVZK proof system, then p_1 does not share $T_{\text{k-d}}$. Next, to obtain the remaining input labels $\mathbf{k}_{in}^e$ of the evaluator p_2, p_1 and p_2 interact with the OT_2^1 scheme defined in Section A.5.3. Initially both parties call the transfer functions. Next, p_1 sends input wire keys encrypted by **ot.Encrypt** as messages ($m_1 = \hat{k}_{\text{in}}^e$, $m_2 = \hat{k}_{\text{in}}^{\neg e}$) to p_2. Party p_2 obtains labels k_{in}^e by calling **ot.Decrypt**. Then, p_2 calls **gc.Evaluate** and if $T_{\text{k-d}}$ has been shared, decodes output wire keys to obtain the output data bit string d_{out}.

A.5.5 Maliciously Secure TwoPC based on dual-execution. We consider running the semi-honest 2PC protocol based on boolean garbled circuits [55] to instantiate the maliciously secure 2PC scheme of the work [27]. Again, the 2PC dual-execution protocol runs two instances of the semi-honest 2PC, where both parties p_1 and p_2 successively act as the garbler and evaluator. Before any 2PC output is shared with the counterparty, the protocol runs a secure validation phase on obtained outputs. The idea of the mutual output verification is as follows. If p_1, as the evaluator, obtains output wire keys $\mathbf{k}_x$ and output bits **b** from a correctly garbled circuit of p_2, then p_1 knows which output labels $\mathbf{k}_y$ according to **b** p_2 must evaluate on a correctly garbled circuit of p_1. Thus, if p_1 shares a commitment in form of a hash $H(\mathbf{k}_y||\mathbf{k}_x)$ with p_2 after the first circuit evaluation, and p_2 returns the same hash $H(\mathbf{k}_y||\mathbf{k}_x)$ after the second circuit evaluation, then p_1 is convinced of a correct garbling by p_2. Because, if p_2 incorrectly garbles a circuit, then p_1 obtains the bits **b**'. And, if p_1 correctly garbles a circuit, p_2 obtains correct bits **b**. The incorrect bits **b**' lead p_1 to a selection of labels $\mathbf{k}'_x$ and $\mathbf{k}'_y$ and the correct bits **b** lead p_2 to a correct selection of $\mathbf{k}_y \neq \mathbf{k}'_y$. Since p_2 does not know which output keys p_1 evaluates, p_2 cannot predict any keys $\mathbf{k}'_x, \mathbf{k}'_y$ which lead to the hash that is expected by p_1. To communicate the output of a maliciously secure 2PC to a single party, only the first garbler is required to share the output decoding table with the counterparty.

Notice: Secure 2PC using the dual-execution mode is susceptible to a single bit leakage. However, if the input is random and leaking even the most significant bit does not yield any advantage, then using this technique is sufficient. The key derivation functions in TLS oracles (e.g. 2PC SHTS circuit) depend on sufficiently random inputs with secret shared input parameters. Thus, applying the dual-execution mode in TLS oracles is feasible [35].

A.6 Zero-knowledge Proof Systems

In practice, zero-knowledge proof systems are implemented by a tuple of algorithms, where

- **zk.Setup**(1^λ, C)⟶ (CRS_C) takes in a security parameter and algorithm, and yields a common reference string,
- **zk.Prove**(CRS_C, x, w)⟶ (π) consumes the CRS, public input x, and the private witness w and outputs a proof π.
- **zk.Verify**(CRS_C, x, π)⟶ $\{0, 1\}$ yields true (1) or false (0) upon verifying the proof π against public input x.

The tuple of algorithms achieves the properties of a zero-knowledge proof systems. If zero-knowledge proof frameworks depend on cryptographic constructions that require a trusted setup (e.g. use pairings or KZG commitments), the **zk.Setup** function must be called by a trusted third party. For transparent instantiations of zero-knowledge proof frameworks (e.g. based on FRI commitments), the **zk.Setup** function can be called by either party. The function **zk.Prove** and **zk.Verify** are called by the prover and verifier respectively.

A.6.1 Zero-Knowledge Succinct Non-Interactive Argument of Knowledge.

A zkSNARK proof system is a zero-knowledge proof system, where the four properties of *succinctness*, *non-interactivity*, computational sound *arguments*, and witness *knowledge* hold [44]. Succinctness guarantees that the proof system provides short proof sizes and fast verification times even for lengthy computations. If non-interactivity holds (e.g., via the Fiat-Shamir security [7]), then the prover is able to convince the verifier by sending a single message. Computational sound arguments guarantee soundness in the zkSNARK system if provers are computationally bounded. Last, the knowledge property ensures that provers must know a witness in order to construct a proof.

A.7 Secret Sharing

We formally define a secret sharing scheme with the following tuple of algorithms, where

- **ss.Setup**(λ)⟶ (pp) takes in a security parameter and returns public parameters and randomness $r \xleftarrow{\$} \mathcal{R}(\lambda)$.
- **ss.Share**(pp, r)⟶ (**r**) takes in public parameters and randomness and returns additive secret shares $\mathbf{r}$=[r_1,. . ., r_n], where $\sum_{x=1}^{n} r_x = r$ holds.
- **ss.Reconstruct**(**r**)⟶ (r) takes in additive secret shares and returns their sum.

A.8 Cryptographic Commitment Schemes

We formally define cryptographic commitments with the following tuple of algorithms, where

- **c.Commit**(m, r_c)⟶ (c) takes in a string m and commit randomness $r_c \xleftarrow{\$} \mathcal{R}$ and yields a commitment string c.
- **c.Open**(m, r_c, c)⟶ ($\{0, 1\}$) takes in a message string, a commit randomness, and a commitment string and outputs 1 only if c is a valid commitment string of the tuple (m, r_c).

The algorithms **c.Commit**, **c.Open** satisfy the properties of a secure commitment scheme, where computational *binding* ensures that after committing to m_1, a probabilistic polynomial time (PPT) adversary cannot find **c.Commit**(m_2, r_2)==**c.Commit**(m_1, r_1), with $(m_1, m_2) \in \mathcal{M}$, $(r_1, r_2) \in \mathcal{R}$, and $m_2 \neq m_1$. Further, anyone seeing c learns nothing on m due to the property of statistical *hiding*, where **c.Commit**(m_1, r_c) is statistically indistinguishable from **c.Commit**(m_2, r_c) with $(m_1, m_2) \in \mathcal{M}$ and $r_c \in \mathcal{R}$.

B Security Analysis

The security analysis concerns the deployment of the HVZK proof system and the unilateral validation in the asymmetric privacy setting. Further, we show that the *Janus* protocol is secure against malicious adversaries during the mutual authentication of the SHTS parameter. The security analysis relies on our threat and system model (cf. Section 3) and uses our formalized cryptographic building blocks (cf. Sections 2.3, Appendix A)

B.1 Construction 1

The first construction creates a maliciously secure evaluation of the HVZK proof system in the asymmetric privacy setting. The proof system leverages semi-honest 2PC based on boolean garbled circuit [29] and is combined with a unilateral validation phase. To show the security of the construction, we first define the security guarantees of the asymmetric privacy setting and conclude that the unilateral validation protocol patches remaining vulnerabilities.

Theorem 1. *If three parties p_0, p_1, and p_2 with access to*

- *a three-party TLS handshake protocol Π_{3PHS}*
- *a secure commitment scheme Π_{com}*
- *a secret sharing scheme Π_{ss} with p_0 as the trusted dealer*
- *a secure channel $sc_{0\text{-}1}$ between p_0 and p_1*
- *a secure channel $sc_{1\text{-}2}$ between p_1 and p_2*
- *a secure channel $sc_{0\text{-}2}$ between p_0 and p_2*
- *a maliciously secure 2PC scheme Π_{2PC} between p_1 and p_2*

perform the sequence of computations

1. *p_0 calls [r_1, r_2]= Π_{ss}.Share(r), with $r \xleftarrow{\$} \mathcal{R}(\lambda)$*
2. *p_0 shares r_1 using sc_{0-1} and r_2 using sc_{0-2}*
3. *either p_0 calls c=Π_{com}.Commit(m, r) with bit strings m, c and shares m, c using sc_{0-1} and c using sc_{0-2}, or Π_{2PC} evaluates Π_{com}.Commit(m, r_1+r_2) where p_1 has m*

(4) p_2 shares r_2 using sc_{1-2}

under the assumptions that

- *the TLS 3PHS implements Π_{ss} and the sequence of computations (1) and (2)*
- *p_0 discards calling Π_{com}.Open*
- *p_0 cannot be compromised by the adversary*
- *p_1 never discloses the secret share r_1*
- *the security of the schemes Π_{2PC}, Π_{3PHS}, etc. holds (e.g. 3PHS relies on the discrete logarithm hardness to find a from aG, with random $a \xleftarrow{\$} EC(\mathbb{F}_p)$ and base point $G \in EC(\mathbb{F}_p)$)*

we say that asymmetric privacy holds between p_1 and p_2 such that only p_1 can call Π_{com}.Open.

Proof 1.1: The security of the 3PHS keeps secret shares confidential. Without access to the initially shared secret shares, the adversary $\mathcal{A}$ cannot compute the commitment string c. Further, the security of the commitment scheme prevents the adversary from finding a collision of c. When computing the commitment through a maliciously secure 2PC system, then $\mathcal{A}$ cannot learn any information on the inputs of the counterparty. Since all parties use secure channels to communicate parameters, $\mathcal{A}$ learns nothing of communicated parameters. Thus, $\mathcal{A}$ cannot find any m or reconstruct r which prevents $\mathcal{A}$ from calling Π_{com}.Open.

Theorem 2. *If two parties p_1 and p_2 with access to*

- *a HVZK proof system Π_{HVZK} using a semi-honest 2PC system Π_{sh2PC}*
- *two secure commitment scheme Π^1_{com}, Π^2_{com}*
- *an asymmetric privacy setting Π_{asym} using Π^2_{com}*
- *a 2PC circuit C_{open} implementing Π^2_{com}.Open*
- *a secure channel $sc_{1\text{-}2}$ between p_1 and p_2*
- *a unilateral validation Π_{uv} using Π^2_{com}*

perform the sequence of computations

(1) Π_{HVZK}.Setup: p_2 calls p=Π_{sh2PC}.Garble(C_{open})
(2) Π_{HVZK}.Setup: p_2 shares $\{p \setminus T_{k-d}\}$ using $sc_{1\text{-}2}$
(3) Π_{HVZK}.Prove: p_1 calls k=Π_{sh2PC}.Evaluate
(4) Π_{uv}: p_1 calls c=Π^1_{com}.Commit(k,r) with $r \xleftarrow{\$} \mathcal{R}(\lambda)$
(5) Π_{uv}: p_1 shares c using $sc_{1\text{-}2}$
(6) Π_{uv}: p_2 shares $\{p\}$ using $sc_{1\text{-}2}$
(7) Π_{uv}: p_1 recomputes C_{open} to verify $\{p\}$
(8) Π_{uv}: p_1 shares r using $sc_{1\text{-}2}$
(9) Π_{HVZK}.Verify: p_2 calls Π^1_{com}.Open(c,r)

under the assumptions that

- *in Π_{sh2PC} p_1 acts as the evaluator and p_2 acts as the garbler*
- *Π_{asym} gives p_1 access to Π^2_{com}.Commit*

we say that after running Π_{asym}, composition of Π_{HVZK} and Π_{uv} as Π_{comp} establishes security against malicious adversaries.

Proof 1.2: The security of Π_{sh2PC} allows the adversary $\mathcal{A}$ to maliciously garble the circuit C_{open}. However, if $\mathcal{A}$ receives c upon disclosure of {p \ T_{k-d}}, the hiding property of Π_{com} prevents $\mathcal{A}$ from learning any secret information on the 2PC inputs of p_1. Further, p_1 detects a cheating $\mathcal{A}$ at the sequence number (7) and aborts the protocol before disclosing r to $\mathcal{A}$. Further, Π_{sh2PC} prevents $\mathcal{A}$ from predicting a k that corresponds to a 1. If $\mathcal{A}$ uses Π^1_{com} to commit garbage, then p_2 aborts at the sequence number (9).

Notice. We define Π_{comp}(Π_{sh2PC}=arg1, C_{open}=arg2, Π^2_{com}=arg3) as an construction that takes as input a semi-honest 2PC system which is executed in the context of the HVZK proof system. The HVZK proof system evaluates a 2PC circuit as the second argument. The third argument is a commitment scheme which establishes the asymmetric privacy setting.

B.2 Construction 2

The second construction provides the *verifier* with a secure authenticity verification of the TLS 1.3 SHTS secret in a setting with malicious adversaries. To do so, the construction combines the effects of the TLS 1.3 1-RTT mode with the TLS 3PHS and a secure 2PC computation of the session secret SHTS. This combination introduces an unsolvable challenge to the adversary which prevents the adversary from forging the authenticity of SHTS.

Theorem 3. *If three parties p_0, p_1, and p_2 with access to*

- *a secure channel $sc_{0\text{-}1}$ between p_0 and p_1*
- *a secure channel $sc_{0\text{-}2}$ between p_0 and p_2*
- *a three-party TLS handshake protocol Π_{3PHS}*
- *a secure commitment scheme Π_{com}*
- *a maliciously secure 2PC scheme Π_{2PC} between p_1 and p_2*
- *a secret sharing scheme Π_{ss} with p_0 as the trusted dealer*
- *a secure AEAD scheme Π_{AEAD}*
- *a secure signature scheme Π_σ where p_0 maintains the private key sk*

perform the sequence of computations

(1) p_0 calls [r_1, r_2]= Π_{ss}.Share(r), with $r \xleftarrow{\$} \mathcal{R}(\lambda)$
(2) p_0 shares r_1 using sc_{0-1} and r_2 using sc_{0-2}
(3) p_2 samples $t \xleftarrow{\$} \mathcal{R}(\lambda)$ and discloses t
(4) p_0 calls c=Π_{com}.Commit(t, r), with bit strings c
(5) p_0 calls σ=Π_σ.Sign(sk, t)
(6) p_0 calls s=Π_{AEAD}.Seal(c,σ) and discloses s
(7) Π_{2PC} evaluates Π_{com}.Commit(t, r_1+r_2)
(8) p_2 calls σ=Π_{AEAD}.Open(c,s) and checks Π_σ.Verify(pk,t, σ)

under the assumptions that

- *the TLS 3PHS implements Π_{ss} and the sequence of computations (1) and (2)*
- *p_0 cannot be compromised by the adversary*
- *pk, and t are public*
- *p_0 never discloses sk*
- *p_2 only performs step (7) if a s has been captured*

we say that an PPT adversary has negligible probability with respect to λ in forging c such that p_2 accepts step (8) and that c is authentic.

Proof 2.1: Again, Π_{3PHS} and Π_{2PC} keep the secret shares confidential. Thus, the adversary $\mathcal{A}$ can only access c at step (7). With c, the adversary can forge a new transcript s but cannot change a s which has already been captured by p_2. Thus, the challenge for $\mathcal{A}$ is to predict a valid c' at a point in time where c remains hidden. Predicting a correct c requires $\mathcal{A}$ either to find a collision for c which the secure commitment prevents. Or, $\mathcal{A}$ correctly guesses the secret share r_2 which evaluates to a correct c before a s is captured by p_2. In the case of a correct guess, $\mathcal{A}$ can replay a σ' on previous t' and encrypt σ' under the right c such that p_2 accepts. However, guessing r_2 or r_1 has negligible probability in λ.

B.3 Construction 3

The third construction reduces the security requirements of cryptographic constructions in the garble-then-prove paradigm [53]. Specifically, we show that the existence of a computation trace to an authenticated commitment string to allows to replace a semi-honest 2PC system based on authenticated garbling with a semi-honest 2PC system that does not require authenticated garbling. Our garble-then-prove paradigm leverages the efficient proof system construction in the asymmetric privacy setting in the prove phase. Further it requires commitment authenticity through SHTS. Thus, for this construction, we use our definitions of Π_{comp} and Π_{auth} (cf. proof 1.1, 1.2, and 2.1 of Appendix B).

Theorem 4. *If two parties p_1 and p_2 with access to*

- *a garble-then-prove scheme $\Pi_{g\text{-}t\text{-}p}$ using two semi-honest 2PC system Π^1_{sh2PC} , Π^2_{sh2PC}*
- *a composition scheme Π_{comp}*
- *a secure commitment scheme Π_{com}*
- *an authenticated commitment scheme Π_{auth} using Π_{com}*
- *a 2PC circuit C_{open} implementing $\Pi_{com}.Open$*
- *a 2PC circuit $C_{kdc+record}$ implementing the TLS 1.3 specification*
- *a 2PC circuit C_{ϕ} implementing a data compliance check against a statement ϕ*

perform the sequence of computations

1. *$\Pi_{g\text{-}t\text{-}p}$.Garble: p_1 calls Π^1_{sh2PC}.Garble($C_{kdc+record}$)*
2. *$\Pi_{g\text{-}t\text{-}p}$.Garble: p_2 calls Π^1_{sh2PC}.Evaluate($C_{kdc+record}$)*
3. *$\Pi_{g\text{-}t\text{-}p}$.Prove: $\Pi_{comp}(\Pi^2_{sh2PC}$, $(C_{kdc+record} + C_{open} + C_{\phi})$, $\Pi_{com})$*

under the assumptions that

- *in Π^1_{sh2PC} p_2 acts as the evaluator and p_1 acts as the garbler*
- *in Π^2_{sh2PC} p_1 acts as the evaluator and p_2 acts as the garbler*
- *Π_{auth} initially authenticates Π_{com}*

we say that malicious security holds for the garble-then-prove paradigm with a semi-honest 2PC system in the garble phase.

Proof 3.1: The adversary $\mathcal{A}$ is able to maliciously garble Π^1_{sh2PC} and obtain secrets from p_2. However, due to the asymmetric privacy setting established during the prove phase, $\mathcal{A}$ learns nothing beyond what $\mathcal{A}$ would have learned during the prove phase. And, a malicious garbling of $\mathcal{A}$ is recorded at p_2 because p_2 obtains all outputs of 2PC circuits executed in the garble phase. Thus, once the construction proceeds to step (3), and $\mathcal{A}$ has cheated, p_2 is able to detect it in step (9) of the Π_{comp} construction and can abort the protocol. This conditional abort option prevents $\mathcal{A}$ from obtaining a false provenance attestation of TLS data.

C Benchmarks Extended

The following subsection provides additional benchmarks.

C.1 Cipher Suite Analysis

To evaluate cipher suite support among today's APIs, we scanned the first 15k entries of the `top-1m.csv.zip` list[11]. To perform the scan, we rely on a publicly available TLS cipher suite scanner[12]. We remove scans which encounter network errors (e.g. no such host) or TLS errors (e.g. EOF, handshake failures). The cipher suite support distribution is depicted in Figure 13. TLS 1.2 configured with GCM reaches a support of 73.5% while TLS 1.2 CBC-HMAC reaches 70.05%. TLS 1.3 support is lower at 55.8%. Even though TLS oracles relying on CBC-HMAC have efficient record phase computations, multiple attacks on the CBC MAC-then-encrypt pattern have been introduced [2, 28, 33]. Even though countermeasures exist, protecting records with the CBC MAC-then-encrypt pattern is not recommended anymore [48]. Our distribution of scans aligns with this recommendation. Further, most endpoints support AEAD cipher suites, where the TLS 1.2 support is 17.7% ahead of TLS 1.3.

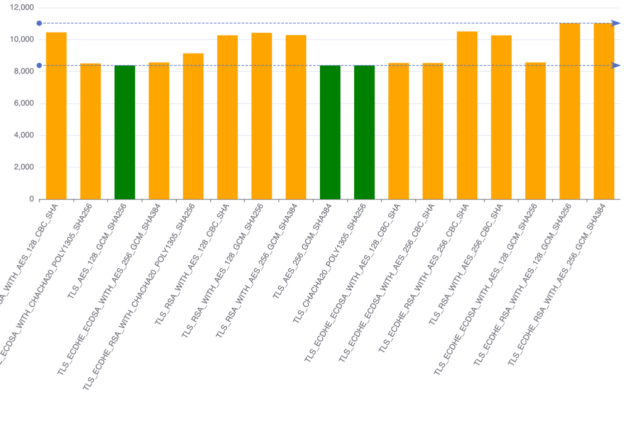


Figure 13: TLS cipher suite scan performed at the 11th of June 2024. Green bars refer to TLS 1.3 cipher suites and yellow bars indicate TLS 1.2 cipher suites.

Table 6: Maliciously-secure 2PC benchmarks averaged over 10 executions using the framework *emp-ag2pc*.

2PC Functions	Communication		Execution	
	Online	Offline	Online	Offline
SHA256	13.184 KB	12.16 MB	10.095 ms	59.4 ms
AES128	6.8 KB	4.33 MB	2.8 ms	55.5 ms

C.2 Microbenchmarks of 2PC Circuits

We present micro benchmarks of secure computation building blocks in Tables 6 and 7. Table 7 compares circuit complexities, execution times, and communication overhead of 2PC circuits, where execution times and communication overhead is further divided into offline and online benchmarks. The 2PC circuits C_{XHTS} and $C_{\text{k,iv}}$ derive session secrets in milliseconds and compute CBs via the circuit $C^{\text{X}}_{\text{CB}_{2+}}$ for a 2 kB record in 164.9 milliseconds. An interesting fact to notice is that the AEAD tag circuit C_{tag} is efficient for small request sizes and scales sufficiently but not ideally for larger request sizes. The overhead in the circuit C_{tag} is introduced by the algebraic structure of the Galois field polynomials in GF(2^{128}), which, as an algebraic structure, is in conflict with the binary representation of computation in boolean GCs. The related works [10, 43] propose a scalable OT-based computation of the AEAD tag, which we consider as future work to improve our implementation.

[11] https://github.com/PeterDaveHello/top-1m-domains
[12] https://github.com/TeoLj/TLSscanner

Table 7: Secure computation benchmarks to implement the *Janus* optimizations. We allocate values to offline/online execution and communication columns. Dashed lines indicate which circuits belong to the handshake, record, and post-record phases.

2PC Circuit	Constraints (x10^6)	Execution Offline	Execution Online	Communication Offline	Communication Online
ECTF	-	-	212.96 ms	-	1.861 kB
C_{XHTS}	3.14	215.56 ms	144 ms	34 MB	110 kB
$C_{\text{k}^{m_1},\text{iv}}$	10.34	723.96 ms	484.82 ms	108.08 MB	356 kB
$C^{256\text{ B}}_{\text{ECB}_{2+}}$ / $C^{2\text{ kB}}_{\text{ECB}_{2+}}$	1.16 / 9.18	67.78 / 578.76 ms	67.6 / 164.9 ms	10.12 / 86.02 MB	116 / 566 kB
$C^{256\text{ B}}_{tag}$ / $C^{2\text{ kB}}_{tag}$	4.04 / 29.01	285.98 ms / 2.42 s	492.24 ms / 3.78 s	52.06 / 378.02 MB	512 kB / 2 MB
$C^{256\text{ B q, 2 kB r}}_{\text{tpOpen}}$	12.69	0.89 s	0.46 s	126.01 MB	583 kB
$C^{256\text{ B q, 2 kB r}}_{\text{zkOpen}}$ / f_ϕ	12.73 / 17.15	0.89 / 1.13 s	2.04 / 2.08 s	127.02 / 168.03 MB	2.13 / 2 MB

Concerning data opening times, we can see that the *transparent* mode with the circuit C_{tpOpen} is more efficient compared to the *privacy-preserving* mode with the circuit C_{zkOpen}. This behavior is expected because, the 2PC circuit of the transparent mode does not include the ciphertext, SHTS, and CB$_{tag}$ verification inside the circuit (cf. Figure 8). As a consequence, the data communicated in the OT scheme of the *transparent* mode is about half the size of the *privacy-preserving* mode. The effect is further visible in the online communication cost, where the *transparent* mode communicates 3x less data than the *privacy-preserving* opening mode. As another reference benchmark (cf. f_ϕ of the last row in Table 7), we evaluate the verification of a confidential document hash $H(f)$ in the circuit C_{zkOpen}. To do so, we set the function f_ϕ=$H(f) \overset{?}{=} H(\mathbf{pt})$ to a hash check on the 2 kB response data, with H=SHA256. Concerning online execution times, the extra hash evaluation yields a negligible overhead for the *client* but increases the communication overhead by a factor of 1.3x.

D Complete E2E Janus Protocol

To simplify the reproduction of this work, we provide complete E2E descriptions of TLS oracles integrating the *Janus* optimizations. We consider a simple scenario where a single request and response is exchanged during the TLS session.

Concerning TLS 1.2, we describe a TLS oracle configured with TLS MtE (e.g. CBC-HMAC). This configuration performs best due to the constant-size 2PC circuits of the record phase (cf. Section 4.1.2). We depict the full E2E protocol of a TLS 1.2 oracle using the first *Janus* contribution in Figure 14. TLS 1.3 oracles are configured with AEAD cipher suites. We depict a full E2E protocol of a TLS 1.3 oracle using both *Janus* optimizations in Figure 15. Notice that for the computation of $C_{\mathbf{zkOpen}}$, the key shares kv, kx indicate the secrets s_1, s_2 of the circuit (cf. Figure 8).

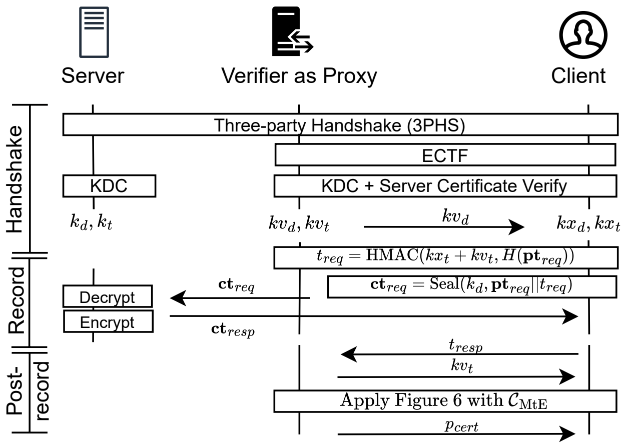

Server
Verifier as Proxy
Client
Handshake
Record
Post-record
Three-party Handshake (3PHS)
ECTF
KDC
KDC + Server Certificate Verify
k_d, k_t
kv_d, kv_t
kv_d
kx_d, kx_t
$t_{req} = \text{HMAC}(kx_t + kv_t, H(\mathbf{pt}_{req}))$
$\mathbf{ct}_{req} = \text{Seal}(k_d, \mathbf{pt}_{req}||t_{req})$
$\mathbf{ct}_{req}$
Decrypt
Encrypt
$\mathbf{ct}_{resp}$
t_{resp}
kv_t
Apply Figure 6 with $\mathcal{C}_{\text{MtE}}$
p_{cert}

Figure 14: End-to-end protocol of a TLS 1.2 oracle running the first *Janus* optimization. Boxes crossing two vertical lines indicate 2PC protocols.

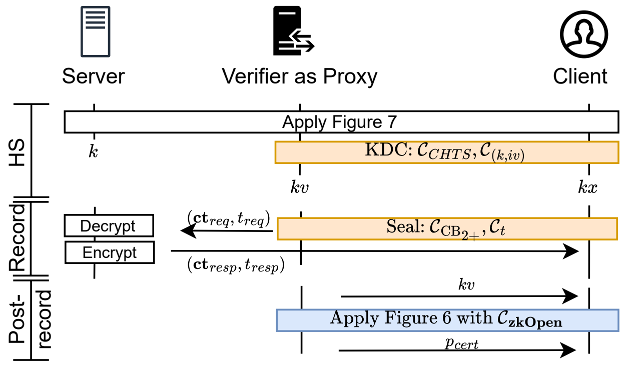

Server
Verifier as Proxy
Client
HS
Record
Post-record
Apply Figure 7
k
KDC: $\mathcal{C}_{CHTS}, \mathcal{C}_{(k,iv)}$
kv
kx
$(\mathbf{ct}_{req}, t_{req})$
Seal: $\mathcal{C}_{\text{CB}_{2+}}, \mathcal{C}_t$
Decrypt
Encrypt
$(\mathbf{ct}_{resp}, t_{resp})$
kv
Apply Figure 6 with $\mathcal{C}_{\mathbf{zkOpen}}$
p_{cert}

Figure 15: End-to-end protocol of a TLS 1.3 oracle running both *Janus* optimizations. Boxes crossing two vertical lines indicate 2PC protocols. Orange boxes belong to the garble phase and the blue box indicates the prove phase.