Janus: Fast Privacy-Preserving Data Provenance For TLS

Jan Lauinger, Jens Ernstberger, Andreas Finkenzeller, Sebastian Steinhorst
Technical University of Munich
Munich, Germany

***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 are practical in verifying private data up to 1 kB in size selectively, which limits their applicability to larger sensitive data sets. In this work, we introduce a new oracle protocol for TLS, which reaches new scales in selectively verifying the provenance of confidential web data. The novelty of our work is a construction which combines an honest verifier zero-knowledge proof system with a new secure validation phase tailored to an asymmetric privacy setting between collaborative TLS clients. Compared to previous works, our construction proves non-algebraic TLS algorithms faster while retaining equivalent security properties. Concerning TLS 1.3, we optimize end-to-end performances and show how the garble-then-prove paradigm can benefit from previously established authenticity to employ semi-honest secure computations without authentic garbling. Our performance improvements show that 8 kB of sensitive TLS data can be verified in 6.7 seconds, outperforming related works significantly. With that, we enable new boundaries to verify the provenance of confidential documents of the web.**

***Index Terms*—Data Provenance, Zero-knowledge Proofs, Secure Two-party Computation, Transport Layer Security**

I. Introduction

Motivation: 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 [1], [2]. 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.

To save a third party from individually verifying data provenance, current approaches either require *servers* to attest to TLS data via digital signatures [3], or employ TLS oracles [4]–[6]. Data attestation through *servers* is an efficient

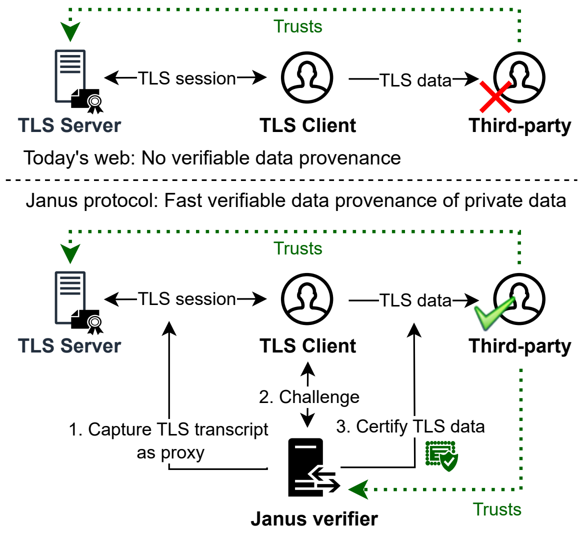


Fig. 1. Illustration of TLS sessions in today's web (top part) and TLS sessions accompanied by a TLS oracle (bottom part). TLS sessions, per default, are secure channels between two parties and prevent a third-party from verifying the provenance of TLS data. In contrast, TLS oracles use a trusted verifier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

data provenance solution but requires server-side software changes and access to a certification infrastructure. In contrast, TLS oracles retain *servers* from the overheads of maintaining a data provenance infrastructure and can be used to entirely outsource the provision 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 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 [4], [6], [7]. However, TLS oracles are generally applicable in the Internet, which makes them a crucial technique to build user-centric and data-sovereign systems [8]. For instance, through TLS oracles, users are able to present solvency checks without giving up control and privacy of their data [9]. Further, data provenance systems enhance electronic surveillance by providing verifiable accountability of confidential web data [10] and can attest if a digital resource originated from a generative AI website [11].

Challenges: Even though different solutions exist, TLS oracles remain constrained in the amount of sensitive data they can validate. This means that larger sensitive resources such as confidential documents, images, or data sets lack the support of verifiable data provenance. For instance, in the work [6], clients are required to prove non-algebraic encryption algorithms (e.g. AES128) in zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) proof systems. But, current zkSNARK proof systems efficiently verify security algorithms that leverage algebraic structures (e.g. MiMC [12]). We measure that proving AES128 on 1 kB of sensitive TLS data using the zkSNARK proof system *Groth16* takes 29 seconds, where *Groth16* introduces a trusted setup security assumption. The work [5] leverages the structure of TLS 1.3 stream ciphers and separates non-algebraic algorithms from the computations performed by the zkSNARK proof system. However, in [5], the *client* is required to know the structure of TLS data in advance and cannot selectively verify dedicated parts of TLS records. Similar to [5], the work [13] shifts the computation of non-algebraic algorithms into a pre-computation phase to improve end-to-end efficiency. Our work addresses the above mentioned limitations with two new contributions. The first contribution optimizes prove computation times during the client challenge (cf. stage 2 in Figure 1) and our second contribution optimizes end-to-end performances of TLS 1.3 oracles.

Contributions: In contrast to the above mentioned works, we leverage the fact that, at some point, TLS oracles introduce an asymmetric privacy setting between collaboratively acting parties; the TLS *client* and the *verifier*. We exploit the asymmetric privacy setting to combine a honest verifier zero-knowledge (HVZK) proof system with a new secure validation phase. The new validation phase is unilaterally performed by the *client* and establishes security guarantees equivalent to related works (e.g. security against malicious adversaries). The HVZK proof system [14] used in our work efficiently evaluates non-algebraic algorithms and proves AES128 on 1 kB of sensitive data in 0.76 seconds. Our approach works transparently and does not require the extra security assumption of a trusted setup. With that, our work achieves a new level of end-to-end efficiency and solves the main bottleneck of current TLS oracles, which is the efficient evaluation of non-algebraic algorithms without compromising on security guarantees. Our first contribution is applicable to TLS oracles that rely on the TLS versions 1.2 and 1.3.

Our second contribution is applicable to TLS 1.3 only and and combines a specific operation mode of TLS 1.3 with the three-party handshake (3PHS) [4], [6], where the *verifier* and TLS client collaboratively operate a secret-shared TLS client. We rely on the TLS 1.3 operation mode where the client hello (CH) message contains a key share that successfully complies with a server-supported cipher suite. Predicting a server-supported cipher suite is an plausible assumption as the TLS client can fetch server-supported TLS ciphers before the establishment of a TLS session. We show that the effects of running TLS 1.3 under such specific circumstances can be used to securely authenticate TLS handshake secrets at the *verifier*. With access to an authentic server handshake traffic secret (SHTS), we optimize the garble-then-prove paradigm of the work [15]. In the garble phase, instead of relying on semi-honest two party computation (2PC) based on authenticated garbling, we rely on semi-honest 2PC systems that do not require authenticated garbling. Our construction is possible because, in the prove phase, we detect malicious activities by recomputing the the garble phase against the authenticity guarantees provided by SHTS. As such, we can take advantage of the performance benefits gained by deploying a more lightweight semi-honest 2PC system.

Results: We compare our optimized proof computation benchmarks for TLS 1.2 and TLS 1.3 oracles against our re-implementations of the works [5], [6]. We use the transparent zkSNARK proof system *PlonkFRI* in our re-implementations to establish security assumptions equivalent to assumptions achieved in this work. As such, we outperform the work [6] by a factor of $382x$ and the work [5] by a factor of $152x$. With our optimized end-to-end performances for TLS 1.3 oracles, we verify 8 kB of public TLS data in 0.58 seconds and verify 8 kB of sensitive TLS data 6.7 seconds.

In analogy to Roman mythology, we name our efficient oracle solutions after the god of transitions, Janus. Because, the *Janus* protocol guards the transition of larger 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 with a new unilateral validation phase.
- We introduce a new TLS 1.3 oracle protocol called *Janus*, which optimized end-to-end efficiency based on an optimized garble-then-prove scheme while retaining security properties equivalent to previous works.
- We analyse the security of our constructions (cf. Appendix C), provide performance benchmarks (cf. Section VI), and open-source[1] the implementation of our secure computation building blocks.

II. Preliminaries

This section highlights the key concepts of TLS which data provenance solutions build upon. In addition, we explain necessary cryptographic building blocks and provide extensive

[1]https://github.com/januspaper/submission1/tree/esp

TABLE I
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"
$(\text{k}_{\text{SATS}}, \text{iv}_{\text{SATS}})$ | $(\text{k}_{\text{CATS}}, \text{iv}_{\text{CATS}})$	DeriveTK(s=SATS|CATS) = (**hkdf.exp**(s,"key",H(""),len(k)), **hkdf.exp**(s,"iv",H(""),len(iv)))

details of each cryptographic construction or protocol in the Appendix B.

A. General Notations

The TLS notations of this work are introduced in Section II-B, 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 $\xleftarrow{\$}$, the concatenation of strings with $||$, and the comparison of variables with $\stackrel{?}{=}$. Concerning authenticated encryption with associated data (AEAD) algorithms in the Galois Counter Mode (GCM) mode, the symbol $M_{\mathbf{H}}$ is a Galois field (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.

B. 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. We focus on TLS 1.2 configured with stream ciphers that perform authenticated encryption (e.g. `TLS_ECDHE_ECDSA_AES128_GCM_SHA256`) in the same way as protocols using TLS 1.3. Generally, 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 B and present TLS-specific transcript hashes, labels, and key derivation functions of traffic keys in Table I.

```
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: $(\text{tk}_{\text{CATS}}, \text{iv}_{\text{CATS}}, \text{tk}_{\text{SATS}}, \text{iv}_{\text{SATS}})$ 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. $b$: $(\text{k}_{\text{CHTS}}, \text{iv}_{\text{CHTS}})$ = DeriveTK(CHTS)
8. $b$: $(\text{k}_{\text{SHTS}}, \text{iv}_{\text{SHTS}})$ = DeriveTK(SHTS)
9. $b$: $\text{fk}_S$ = 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($\text{fk}_S$, $H_7$); send SF in $m_{\text{SF}}$
12. $c$: SF' = hmac($\text{fk}_S$, $H_7$); verify SF' $\stackrel{?}{=}$ SF
13. $c$: ds.Verify($pk_S$, $\text{label}_{11}$ || $H_6$, SCV) $\stackrel{?}{=}$ 1
14. $b$: $\text{fk}_C$ = hkdf.exp(CHTS, "finished" || "")
15. $c$: CF = hmac($\text{fk}_C$, $H_9$); send CF in $m_{\text{CF}}$
16. $s$: CF' = hmac($\text{fk}_C$, $H_9$); verify CF' $\stackrel{?}{=}$ 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. $b$: $(\text{k}_{\text{CATS}}, \text{iv}_{\text{CATS}})$ = DeriveTK(CATS)
22. $b$: $(\text{k}_{\text{SATS}}, \text{iv}_{\text{SATS}})$ = DeriveTK(SATS)
```

Fig. 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.

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. A special mode of TLS 1.3 is that if m_{CH} contains the client randomness and key share which complies with a cipher suite supported at the server, then the server immediately derives handshake and record phase traffic secrets. In this mode, TLS 1.3 encrypts all server-side handshake messages.

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 used AEAD based on stream ciphers, TLS 1.2 generates

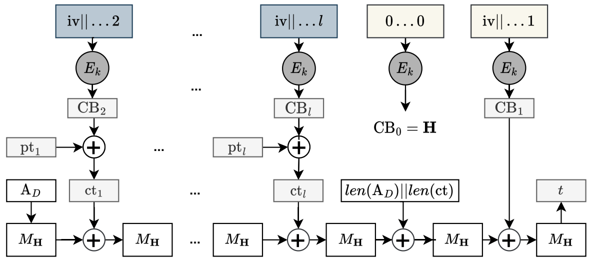
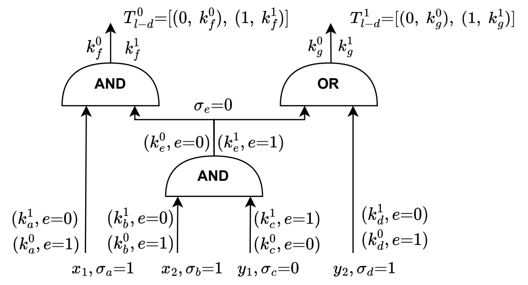


Fig. 3. TLS 1.3 AEAD stream cipher in the GCM mode which encrypts a plaintext $\mathbf{pt} = [pt_1, \dots, pt_l]$ to a ciphertext $\mathbf{ct} = [ct_1, \dots, ct_l]$ under key k and authenticates the ciphertext **ct** and associated data A_D with the tag t.

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 this work, we omit investigating TLS 1.2 in CBC mode as multiple attacks on the CBC MAC-then-encrypt pattern have been introduced [17]–[19]. Even though countermeasures exist, protecting records with the CBC MAC-then-encrypt pattern is not recommended anymore [20]. Per default, TLS 1.3 generates two keys to secure handshake phase traffic (k_{CHTS}, k_{SHTS}) and generates two keys to secure record phase traffic (k_{SATS}, k_{CATS}) Due to a key-independence property of TLS 1.3 [21], disclosing an handshake traffic secret (e.g. SHTS) does not compromise the security of record traffic secrets (e.g. server application traffic secret (SATS)) as access to SHTS prevents the reconstruction of handshake secret (HS) and HS is required to compute SATS.

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) authentication values, because, to constitute a valid TLS session, both parties must successfully exchange and verify the SF and CF messages $m_{\text{SF}}, m_{\text{CF}}$. For 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 [22]. 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) Record Phase: The TLS record phase requires parties to protect data with an AEAD algorithm before data can be exchanged. AEAD algorithms use stream ciphers to protect data and depend on keys established in the handshake phase to translate plaintext data **pt** into a confidential and authenticated representation $(\mathbf{ct}, t)$, with ciphertext **ct** and authentication tag t. Stream ciphers are characterized by pseudorandom generators (advanced encryption standard (AES) in the GCM mode), which incrementally output key streams or counter blocks (CBs) (cf. Figure 3). 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. We elaborate on TLS data protection algorithms in the Appendix B4.



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

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

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

Fig. 4. Example of a garbled circuit $\mathcal{C}$ 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 $\mathbf{e}$=$\sigma_L \oplus \mathbf{i}$ (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 $\boldsymbol{T}_{l-d}$ map output labels to output data bits.

C. Cryptographic Building Blocks

This section provides an overview of the cryptographic fundamentals that support the *Janus* protocol beyond algorithms of TLS. Formal descriptions of these cryptographic building blocks can be found in the Appendix B.

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 [23], [24]. 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 $\boldsymbol{G}(\mathcal{C})$, where the boolean circuit $\mathcal{C}$ implements the logic of the public function f (cf. Figure 4). 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=$\sigma \oplus \mathbf{i}$ which can be shared with p_2. With that, signal bits enable the evaluator to discover valid entries of garbled tables $G(\mathcal{C})$ through external bits e [25]. Further, signal bits randomize garbled truth tables $G(\mathcal{C})$ 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 $G(\mathcal{C})$ (cf. Figure 4), 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 $\boldsymbol{G}(\mathcal{C})$, $\boldsymbol{T}_{l-d}$, and, if **x**=[1,0], $(k_a^1, e{=}0)$ and $(k_b^0, e{=}1)$ 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 II-C2). The OT protocol requires p_1 to share $k_e^{\mathbf{y}}, k_d^{\mathbf{y}}$ with corresponding external values with p_2. Further, the OT scheme gives p_2 access to the keys $(k_c^0, e{=}0)$ and $(k_d^1, e{=}0)$ if **y**=[0,1], and prevents p_1 from learning p_2's selection of wire keys. With access to $\boldsymbol{G}(\mathcal{C})$, 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 $G(\mathcal{C}_{AND}^{0,(1,2)})$ and obtains $(k_e^0, e{=}0)$. With that, p_2 continues to decrypt the first entry of table $G(\mathcal{C}_{AND}^{1,(0,1)})$ to obtain k_f^0 (cf. Figure 4). Last, p_2 decodes k_f^0 using the decoding table T_{l-d}^0 to obtain the first output bit 0. If required, p_2 shares the obtained 2PC output back to p_1.

2) Oblivious Transfer: Secure 2PC based on GCs depends on the 1-out-of-2 OT_2^1 sub protocol to secretly exchange input parameters of the circuit [26]. The OT_2^1 scheme 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. In this work, we make use of the OT_2^1 scheme defined in the work [26], 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.

3) 2PC with Malicious Adversaries: We consider the work [27] to secure the semi-honest 2PC defined in Section II-C1 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. We describe the intuition of behind maliciously secure 2PC in the Appendix B5e.

4) Zero-knowledge 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 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 $x \notin \mathcal{L}$. Proof systems work if the properties of *completeness* and *soundness* hold. Privacy-preserving proof systems additionally require either *zero-knowledge* or *HVZK* to hold [28], [29].

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 [14]. 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 [14].

5) Cryptographic Commitments: We formally define cryptographic commitments with the following tuple of algorithms, where

- **c.Commit**$(m,\ r_c) \rightarrow (c)$ takes in a string m and commit randomness $r_c \overset{\$}{\leftarrow} \mathcal{R}$ and yields a commitment string c.
- **c.Open**$(m,\ r_c,\ c) \rightarrow (\{0, 1\})$ 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 $r_c \in \mathcal{R}$.

Commitment schemes 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.

6) Secret Sharing: We formally define a secret sharing scheme with the following tuple of algorithms, where

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

Secret sharing involves a trusted dealer to break a secret into shares with **ss.Share**. Shares are distributed to qualified recipients which can reconstruct the secret by computing individual shares back together with **ss.Reconstruct** [30]. 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 [31].

III. System Model

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

A. System Roles

Clients establish a TLS session with *servers*, query data from *servers*, and present TLS data proofs to the *proxy*. We assume that *clients* behave maliciously such that *clients* arbitrarily deviate from the protocol specification in order to learn TLS session secret shares of the *proxy*. Another goal of malicious *clients* is to learn any information that contributes to convincing the *proxy* of false statements on presented TLS data. *Clients* honestly follow algorithms of the *Janus* protocol if the algorithm protects private data of the *client*.

Servers participate in TLS sessions with *clients* and return responses in the TLS record phase upon the reception of compliant API queries. We assume honest *servers* which follow the *Janus* protocol specification.

Verifiers act as proxies and take over the role of TLS oracle verifier. *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*. *Verifiers* honestly execute algorithms if algorithms protect secret shares of *verifiers*.

B. System Goals

Session-authenticity guarantees that our TLS oracle attests web traffic which originates from an authentic TLS session. Authenticity is guaranteed if the *proxy* successfully verifies the PKI certificate of the server.

Session-integrity guarantees that a malicious *client* and *proxy* 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 *proxy* 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 *proxy* 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 machine-in-the-middle (MITM) attacks.

Legacy-compatibility holds if the TLS code stack running at the *server* does not require any changes and achieves out-of-the-box compatibility with our protocol.

C. Threat Model

We rely on a threat model with secure communication channels (TLS security guarantees hold) and fresh randomness per TLS session between all interacting system roles. This means that the adversary cannot break the security guarantees of TLS. Network traffic, even if it is intercepted via a 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.

IV. Optimizing Proof Computations in the Asymmetric Privacy Setting

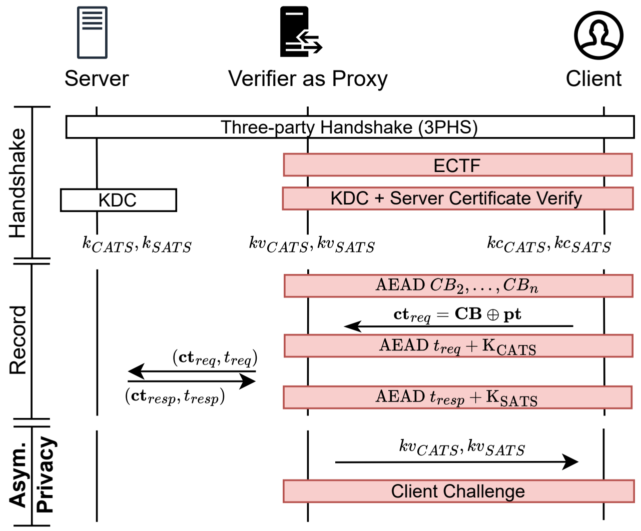
The first subsection IV-A analyses (i) where TLS oracles configured with to use AEAD algorithms introduce an asymmetric privacy setting between the *client* and the *proxy* and (ii) how semi-honest proof systems can be deployed to obtain security against malicious adversaries. In subsection IV-B, we propose a new unilateral validation phase which, combined with an HVZK proof system, achieves security against malicious adversaries (cf. Appendix C1).

A. How TLS Oracles Use Asymmetric Privacy

TLS oracles turn the two-party protocol of TLS into a three-party protocol by introducing a *verifier* [4]. The *verifier* ensures that the TLS data of the *client* preserves integrity according to an authenticated TLS session via a verifiable computation trace.

1) Three-party Handshake: 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 3PHS [4], [6]. In the 3PHS, every party injects a secret randomness such that the DHE secret of the TLS handshake depends on three secrets instead of two. As such, the DHE value, which is derived at the *server*, can be jointly reconstructed if the *client* and *verifier* add shared secrets together. The Appendix B1 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*. At the same time, the *verifier* is convinced that the *client* preserves computational integrity according to the TLS specification if the joint TLS



Server
Verifier as Proxy
Client
Handshake
Three-party Handshake (3PHS)
ECTF
KDC
KDC + Server Certificate Verify
k_{CATS}, k_{SATS}
kv_{CATS}, kv_{SATS}
kc_{CATS}, kc_{SATS}
Record
AEAD $CB_2, \ldots, CB_n$
$\mathbf{ct}_{req} = \mathbf{CB} \oplus \mathbf{pt}$
$(\mathbf{ct}_{req}, t_{req})$
AEAD $t_{req} + \mathrm{K}_{\mathrm{CATS}}$
$(\mathbf{ct}_{resp}, t_{resp})$
AEAD $t_{resp} + \mathrm{K}_{\mathrm{SATS}}$
Asym. Privacy
kv_{CATS}, kv_{SATS}
Client Challenge

Fig. 5. TLS 1.3 oracle adapted from the DECO protocol [6]. After the key derivation computation (KDC), it holds that $k_{SATS} = kv_{SATS} + kc_{SATS}$ and $k_{CATS} = kv_{CATS} + kc_{CATS}$. Algorithms executed by two parties are surrounded by red boxes and achieve security against malicious adversaries.

computations progress. Because, 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*. Further, introducing false session data on the client-side leads to a session abort at the server.

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 [6], [32]. To achieve efficient secure 2PC [4]–[6], TLS oracles convert secret-shared DHE values in form of elliptic curve (EC) coordinates into bit-wise additive secret shares with the Elliptic Curve to Field (ECTF) algorithm (cf. Appendix B5) [33]. Additive secret shares can be efficiently added together in 2PC circuits that are based on boolean GCs [6], [24], [27], [34], [35]. After the ECTF conversion (cf. Figure 5), 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 [30], [36].

3) Key Committing AEAD for TLS Records: AEAD-protected records in TLS require special attention as AES in the GCM mode is not key committing [13], [37]–[39]. This means that an adversary can perform commitment attacks [40]. For instance, the message franking attack finds two messages $m_1 \neq m_2$ 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 [41]. This attack is problematic and would allow the *client* to inject and prove arbitrary data as TLS authentic during the client challenge. The attack fails if the *verifier* receives all ct_{req}, ct_{resp} before computing any authentication tags t using 2PC (cf. Figure 5). Because, to perform the message franking attack, the attacker requires freedom in choosing a ciphertext block based on a linear

```
$\mathcal{C}_{\mathrm{HVZK}}(\mathbf{pt},\ kc\ \ ;\ \ kv,\ \mathbf{ct},\ \mathrm{iv},\ \mathrm{K}_x,\ \phi)$:
1. $\mathbf{CB}' = \mathrm{AES128\_GCM}(kc + kv,\ \mathrm{iv})$
2. $\mathbf{ct}' = \mathbf{CB}' \oplus \mathbf{pt}$
3. assert: $\mathbf{ct}' \overset{?}{=} \mathbf{ct}$; $\mathrm{K}_x \overset{?}{=} f_{\mathrm{com}}(kc + kv$——$,\ \mathrm{iv})$
4. assert: $1 \overset{?}{=} f_\phi(\mathbf{pt})$
5. return: *true*
```

Fig. 6. Circuit logic of the *Janus* protocol proof challenge. The semicolon ; separates private inputs (left side) from public inputs (right side).

equation that involves access to authentication tags [41].

In addition and according to the proxy mode of the work [6], we compute key commitment strings ($\mathrm{K}_{\mathrm{SATS}}$, $\mathrm{K}_{\mathrm{CATS}}$) on application traffic keys. The commitments guarantee that, in the client challenge, a ciphertext can only be decrypted under the right key, and, with that, produce the right plaintext. Similar to the general key commitment construction for AES GCM (cf. $\mathrm{F}_{\mathrm{com}}$(K,N) in [42]), we define the key commitment string based on f_{com} (cf. formula 1), where H is a secure hash function (e.g. SHA256). The key commitment string hides traffic keys with the pseudorandomness generated by AES (32 bytes if AES128 is used) and urges f_{com} to evaluate the SHA256 compression function twice.

$$K_x = f_{\mathrm{com}}(k_x, \mathrm{iv}_x) = H(k_x || \mathrm{AES}_{k_x}(0) || \mathrm{AES}_{k_x}(\mathrm{iv}_x || 1)) \tag{1}$$

As such, the key commitment string of the server traffic key $\mathrm{K}_{\mathrm{SATS}}$ computes as $f_{\mathrm{com}}(k_{\mathrm{SATS}}, \mathrm{iv}_{\mathrm{SATS}})$ and the key commitment string of $\mathrm{K}_{\mathrm{CATS}}$ computes as $f_{\mathrm{com}}(k_{\mathrm{CATS}}, \mathrm{iv}_{\mathrm{CATS}})$.

4) Client Challenge in Asymmetric Privacy Setting: Once the *client* has gathered enough TLS data to receive an attestation from the *verifier*, TLS oracles reveal TLS secret shares of the *verifier* to the *client* (cf. Figure 5). When the *client* obtains full access to session secrets, an asymmetric privacy setting between the *client* and *verifier* is established because the *client* is able to access TLS records by decrypting exchanged ciphertext data. To prevent the *client* from maliciously presenting arbitrary data as TLS data, TLS oracles bind the client challenge against an irreversible condition, which is formulated as follows: If the *verifier* has access to an authenticated ciphertext and the *client* proves that TLS data encrypts to the authenticated ciphertext under valid keys of the TLS session, then *clients* cannot inject arbitrary data and prove it as TLS authentic [6], [38]. Current TLS oracles rely on maliciously secure proof systems to prove that TLS data encrypts to authenticated ciphertext data [6], [13], [32]. However, in subsection IV-B, we show that proof computations in an asymmetric privacy setting can be optimized.

B. HVZK and Asymmetric Privacy

This section formalizes asymmetric privacy and introduces a new unilateral validation protocol that, combined with an HVZK proof system in the asymmetric privacy setting, achieves security against malicious adversaries. Last, we show how our formal definitions apply to TLS.

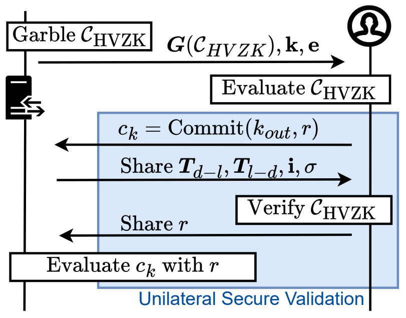

Garble $\mathcal{C}_{\text{HVZK}}$
$\boldsymbol{G}(\mathcal{C}_{HVZK}), \mathbf{k}, \mathbf{e}$
Evaluate $\mathcal{C}_{\text{HVZK}}$
$c_k = \text{Commit}(k_{out}, r)$
Share $\boldsymbol{T}_{d-l}, \boldsymbol{T}_{l-d}, \mathbf{i}, \sigma$
Verify $\mathcal{C}_{\text{HVZK}}$
Share r
Evaluate c_k with r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

Fig. 7. Unilateral validation protocol to assert correct garbling in the HVZK proof system of the work [14].

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. Other settings remain to be investigated as future work. We formalize asymmetric privacy based on the on a maliciously secure 2PC scheme Π_{2PC}, a secure commitment scheme Π_{Com}, and a secret sharing scheme Π_{SS}. We construct asymmetric privacy with formalized algorithms defined in Section II-C.

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 $\mathcal{C}$ in the 2PC scheme Π_{2PC}, where $\mathcal{C}$ 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 shared, p_2 releases the secret share r_2 to p_1, and, with that, initiates the asymmetric privacy setting. In this case, p_1 can reconstruct r. With access to m and r, only p_1 is capable of successfully evaluating Π_{Com}.**Open**.

In 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.

2) HVZK and Selective-failure Attacks: We improve the performance of proof computations in the asymmetric privacy setting by deploying the HVZK proof challenge between p_1 and p_2. The party p_1 has access to all TLS session secrets and we assume p_2 to act as a malicious garbler. The proof system of the work [14] uses semi-honest 2PC based on boolean garbled circuits to achieve the notion of HVZK and assumes an honest *verifier* (cf. Section II-C4). However, in a setting with a maliciously acting p_2, semi-honest 2PC is susceptible to selective failure attacks [34], which work as follows. 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. From here, 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 performed by p_1, which detects malicious behavior of p_2 before any secrecy leakage can occur.

3) Unilateral Secure Validation: The unilateral secure validation is performed once p_1 obtains all semi-honest 2PC parameters of the HVZK proof system [14]. The set of 2PC parameters comprise garbled tables $\boldsymbol{G}(\mathcal{C}_{\text{HVZK}})$, wire keys $\mathbf{k}$, and external labels $\mathbf{e}$ (cf. Section II-C1). The party p_2 shares wire keys corresponding to the private inputs $\mathbf{pt}$ and k of p_1 through the OT_2^1 oblivious transfer protocol [26], and omits sharing the output label decoding table $\boldsymbol{T}_{l-d}$. The garbler p_2 is convinced of the HVZK proof if p_1 as the evaluator returns the output wire key that corresponds the output bit 1. The 2PC circuit $\mathcal{C}_{\text{HVZK}}$ asserts if plaintext chunks encrypt to authenticated ciphertext chunks under the secret shared TLS session keys (cf. Figure 6). Inside $\mathcal{C}_{\text{HVZK}}$, the function f_ϕ continues to assert plaintext data against a public statement ϕ (e.g. ϕ=$\{H_{PDF} \overset{?}{=} H(\mathbf{pt})\}$, where H_{PDF} is a hash of a PDF document with hash function H). The evaluation of the garbled circuit $\mathcal{C}_{\text{HVZK}}$ returns the output wire key corresponding to a 1 if the circuit returns *true*.

The prover p_1 of the proof challenge is required to return the output wire key back to p_2 to complete the HVZK proof protocol. Instead, our new secure validation phase enforces p_1 to share a commitment c_k of the output wire key. After sharing the commitment c_k, 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 $\mathcal{C}_{HVZK}$ 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 of the circuit $\mathcal{C}_{HVZK}$, then p_1 aborts the protocol. Otherwise, p_1 shares the commitment randomness of the commitment c_k with p_2 such that p_2 can verify if p_1 could compute the correct output wire key before the disclosure of the garbling parameters (cf. Figure 7).

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 key commitment strings. Upon capturing a ciphertext ct_x, the *verifier* participates in a joint 2PC computation of the respective authentication tag t_x. Access to authenticated ciphertext chunks 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* if enough ciphertext data has been captured. From here on, only the *client* is capable of computing valid proofs during the client challenge, where valid proofs assert data provenance of TLS data against ciphertext chunks and key commitment strings.

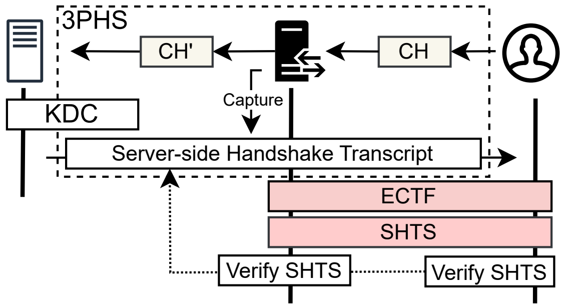


Fig. 8. Mutual SHTS verification between client parties. Red boxes indicate values derived in maliciously secure TwoPC systems.

V. Optimizing End-to-end Performance

The following subsections introduce a new and complete TLS oracle protocol for TLS 1.3 named *Janus*. To construct the protocol, we show how client parties can securely derive and authenticate the TLS handshake secret SHTS in a malicious security setting (cf. Subsection V-A). Subsequently, we leverage the authenticity guarantees of SHTS to deploy a garble-then-prove computation scheme for TLS 1.3, which entirely relies on semi-honest 2PC techniques. Our construction provides the *verifier* with an conditional abort option which activates if the *client* performs malicious activities.

A. Secure Authenticity of SHTS

Our protocol relies on a specific operation mode of TLS 1.3, where the *client* picks a *server*-supported cipher suite and a suitable key share in the CH message. In this case, the *server* immediately derives TLS session secrets and returns authenticated handshake messages (cf. Figure 8). Notice that before establishing the TLS 1.3 session, *clients* can query *servers* to discover supported cipher suites. Upon receiving the server handshake transcript, both the *client* and *verifier* continue to derive shared session secrets via the 3PHS (cf. Appendix B1). Next, the *client* and *verifier* translate shared EC secrets of the 3PHS into additive secret shares s_1, s_2 through the ECTF protocol (cf. Appendix B5b). In the end, the *verifier* locally maintains s_1 and the *client* locally keeps s_2 and it holds that $s_1 + s_2 = \text{DHE}$.

1) Mutual Validation of SHTS: To derive SHTS between clients, the *verifier* garbles the circuit $\mathcal{C}_{\text{SHTS}}$ (cf. Table II) in a semi-honest 2PC system. The *Janus* protocol defines one condition for the evaluation of $\mathcal{C}_{\text{SHTS}}$; Both client parties must obtain the server handshake transcript before participating in the 2PC computation of the circuit $\mathcal{C}_{\text{SHTS}}$ (cf. Figure 8). The condition prevents adversaries from forging the authenticity of SHTS because to compute a valid server handshake transcript, the adversary needs full access to (i) the *server*'s private key corresponding to the X.509 PKI certificate of the server, and to (ii) the session secrets DHE or SHTS. Our system model assumes that the *server*'s private key cannot be compromised and the 3PHS prevents full access to session secrets. The only remaining attack left to the adversary to perform is a replay attack using a previously established server handshake transcript. The replayed transcript must comply with the SHTS value derived in a maliciously secure 2PC system. However, without knowledge of full handshake secrets, the adversary remains incapable of predicting secret 2PC inputs that evaluate to a SHTS collision (cf. Appendix C2).

TABLE II
Semi-honest 2PC circuits used by the *Janus* protocol.

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

Further and similar to the works [6], [38], 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 [21], 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.

The validation of the SHTS parameter against the server handshake transcript works as follows. With access to SHTS, both client parties derive handshake traffic secrets $k_{\text{SHTS}}, \text{iv}_{\text{SHTS}}$ and decrypts server-side handshake messages. With handshake messages in plain, client parties assert if (i) the server certificate is valid, and (ii) the server successfully agreed to the TLS session by verifying the SF message. Notice that access to SHTS allows the computation of the SF verification tag (cf. lines 9-12 in Figure 2). Once the *verifier* completes the SHTS verification, the *Janus* protocol proceeds with the garble-then-prove phase.

B. Garble-then-prove with Semi-honest 2PC

The garble-then-prove scheme of the *Janus* protocol is closely related to the garble-then-prove paradigm introduced in the work [15]. But, in the garble phase, we replace 2PC computations based on authenticated garbling with more lightweight 2PC computations that do not require authenticated garbling. Our construction detects malicious activities in the prove phase, where we recompute and authenticate garbled parameters against the authenticity guarantees provided by the SHTS validation (cf. subsection V-A). We show the security of the construction in the Appendix C3

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* is capable of maliciously garbling any circuit in the garble phase. However, as TLS oracles eventually disclose session secrets of the *verifier* to the *client*, the malicious *client* learns nothing beyond what the honest *client* would have learned. Access to the *verifier*'s secret shares before a protocol-conform asymmetric privacy setting compromises the security of the *Janus* protocol (e.g. enables message franking attacks). However, the prove phase provides guaranteed detection of

```
$\mathcal{C}_{\textbf{zkOpen}}(s_2$, $\mathbf{pt}$ ; $s_1$, $\mathbf{I}^{\text{open}}$, $\mathbf{ct}$, $t$, SHTS, $\phi$):
1. SHTS', $\mathbf{CB}$', $t$' = $\mathcal{C}_{\text{open}}(s_1$, $s_2$, $\mathbf{I}^{\text{open}}$, $\mathbf{pt}$)
2. $\mathbf{ct}$' = $\mathbf{CB}$' $\oplus$ $\mathbf{pt}$
3. assert: SHTS'$\overset{?}{=}$SHTS; $\mathbf{ct}$'$\overset{?}{=}\mathbf{ct}$; $t$'$\overset{?}{=}t$
4. assert: $1\overset{?}{=}f_\phi(\mathbf{pt})$
5. return: true
```

Fig. 9. 2PC circuit for the privacy-preserving client challenge. The circuits assert if the *client* presents data which preserves *session-integrity* and *session-authenticity*. The semicolon **;** separates private inputs (left side) and public inputs (right side). The function f_ϕ validates if input data complies with a public statement ϕ.

cheating activities and allows the *verifier* to abort before any data provenance attestation has occurred. To do so, the prove phase recomputes and compares all 2PC computations of the *client* against securely authenticated session parameters (cf. subsection V-B3).

2) Garble Phase: In the garble phase, the *client* and *verifier* collaboratively evaluate multiple 2PC circuits that implement the TLS 1.3 specification. The set of circuits comprises key derivation computations with $\mathcal{C}_{(\text{k,iv})}$, CB computations for the encryption and decryption of requests or responses with $\mathcal{C}_{\text{CB}_{2+}}$, and tag computations with $\mathcal{C}_t$ (cf. Table II). For the encryption or decryption of requests or responses, the circuit $\mathcal{C}_{\text{CB}_{2+}}$ outputs counter blocks CB_i with indices $i > 1$. To prevent commitment attacks on response records, no block CB_{2+} ever includes any CB_0,CB_1 blocks.

We expect parties to exchange 2PC outputs according to our adapted TLS 1.3 oracle protocol (cf. Figure 5) with slight changes. Computing key commitment strings is redundant because the client challenge in the prove phase considers the authenticated SHTS parameter as the commitment on session keys.

3) Prove Phase: After the garbling phase, the *verifier* has captured a record transcript with the parameters ($\mathbf{ct}_{req}$, t_{req}, $\mathbf{ct}_{resp}$, t_{resp}) and discloses secret-shared session parameters to the *client*. With that, the asymmetric privacy setting is established (cf. Section IV-B). The novelty of the *Janus* protocol is that it can consider the authenticated SHTS parameter as a key commitment string. Thus, the HVZK 2PC circuit $\mathcal{C}_{\text{HVZK}}$=$\mathcal{C}_{\text{HVZK}}$ can be set to the $\mathcal{C}_{\textbf{zkOpen}}$ algorithm, which asserts if application traffic key computes to the authentic SHTS parameter (cf. Figure 9). Further, to ensure that 2PC outputs derived by the *client* have been garbled correctly, the circuit $\mathcal{C}_{\textbf{zkOpen}}$ authenticates ciphertext chunks as follows. A record pair ($\mathbf{ct}$, t) has been garbled correctly if the captured authentication tag matches a recomputed authentication tag t that maps to SHTS. In the prove phase, we require executing the client challenge with an unilateral-validated HVZK proof system to prevent the *verifier* from maliciously garbling the circuit $\mathcal{C}_{\textbf{zkOpen}}$ (cf. Section IV-B).

C. Additional Considerations

The following subsections complete the context of the *Janus* protocol beyond its main contributions from subsections V-A and V-B.

1) Janus Operation Modes: The *Janus* protocol 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 is selected by the *client* and shared with the *verifier* in the beginning of the prove phase. The list $\mathbf{I}^{\text{open}}$ contains indices of counter blocks, ciphertext chunks, and plaintext chunks. With $\mathbf{I}^{\text{open}}$, the *client* selectively determines TLS data for the client challenge. The *verifier* uses $\mathbf{I}^{\text{open}}$ to identify public inputs (e.g. ciphertext substrings) for the data provenance verification (cf. Figure 9).

Transparent Mode If the *Janus* protocol operates in the *transparent* mode, the *client* shares TLS plaintext chunks $\mathbf{pt}$ together with the list of indices $\mathbf{I}^{\text{open}}$ with the *verifier*. The *verifier* checks if the AEAD encryption of presented plaintext yields the captured ciphertext transcripts and if an TLS 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 $\mathcal{C}_{\textbf{tpOpen}}$. $\mathcal{C}_{\textbf{tpOpen}}$ works as $\mathcal{C}_{\textbf{zkOpen}}$ with the exception that the plaintext is used as a public input parameter. Further, the assertion $1 \overset{?}{=} f_\phi(\mathbf{pt})$ can be computed out-of-circuit.

Privacy-preserving Mode If the *Janus* protocol operates 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 $\mathcal{C}_{\textbf{zkOpen}}$ and applies the unilateral secure validation.

Example: We assume that TLS is configured to use AES in the GCM mode. Further, we assume that the TLS data of interest for the validation according to f_ϕ is contained in ct_3 of the response ($\mathbf{ct}$, t). In this case, the index i=3 is included in the list $\mathbf{I}^{\text{open}}$. With $\mathbf{I}^{\text{open}}$, the **zkOpen** circuit is able to compute the right $\text{CB}_{2+\text{index}}$, and consider CB_5=AES$(k, \text{iv}||\ldots 5)$ for the computation of ct_5'=$\text{CB}_5 \oplus pt_5$. If the assertions against public inputs succeed (e.g. ct'_5 ?= ct_5), and CB_5 has been derived with secrets that match a verified SHTS, then the data inside pt_5 preserves *session-integrity* and *session-authenticity*.

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 $\mathcal{C}_{\text{CB}_{2+}}$ is only called for the compilation of request ciphertexts. Response 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 $\mathcal{C}_{\text{CB}_{2+}}$. The evaluation of l circuits $\mathcal{C}_{\text{CB}_{2+}}$ 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 $\mathcal{C}_{\mathrm{CB}_{2+}}$ outputs the corresponding $\mathbf{CB}_{2+}$ of response $\mathbf{ct}$.

3) Optimized HVZK Circuit to Verify AEAD Tags: The computation of authentication tags in the HVZK 2PC circuits can be reduced to two executions of AES, which yield the counter blocks as $\mathbf{CB}_t$=[$\mathrm{CB}_0, \mathrm{CB}_1$] (cf. Figure 3). Deriving the counter blocks $\mathbf{CB}_t$ in-circuit is sufficient because the 2PC invocations of AES hide the TLS application traffic keys. As such, the *verifier* takes $\mathbf{CB}_t$, ciphertext chunks ct, and additional data A_{D} and recomputes the rest final computations of the tag t out of circuit. This optimization disposes the algebraic GF operations during tag computations, which are expensive to compute using boolean logic gates.

4) Data Attestation: If the client challenge succeeds successfully, the *verifier* attest to TLS data validated in the HVZK proof system. The attestation of the *verifier* depends on the *Janus* protocol operation modes. If *Janus* operates in the transparent mode, then the *verifier* hashes verified TLS data of the *client* and signs the hash. The certification parameter p_{cert}=(t, ϕ, pk, σ) of the transparent attestation includes a signature σ=**ds.Sign**(sk,[ϕ, 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. If *Janus* operates in the privacy-preserving mode, the structure of p_{cert} remains the same except that the statement ϕ expresses data compliance constraints as a string. The privacy-preserving attestation convinces any third party of the fact that the *verifier* successfully validated TLS data provenance against the statement ϕ at time t. The certificate p_{cert} enables verifiable data provenance of TLS data as p_{cert} can be verified by any third party who trusts the *verifier*.

VI. 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 end-to-end performance of TLS 1.3 oracles. We provide micro benchmarks on a circuit level in the Appendix A.

A. Implementation

We implemented the 3PHS by modifying the Golang *crypto/tls* standard library[2] and configured the NIST P-256 elliptic curve for the elliptic curve Diffie–Hellman exchange (ECDHE). Our proof of concept implementation configures either TLS version with a cipher suite relying on AES128 in the GCM mode and SHA256 as the hash function. We implement the ECTF conversion algorithm in Golang using the Paillier cryptosystem [43]. For a coherent implementation of the *Janus* protocol in Golang, we chose the *mpc* library [44] to access a semi-honest 2PC system based on garbled circuits and we chose the *gnark* framework [45] to implement ZKP circuits of related works. We adjusted the *mpc* library to output single wire labels if we execute 2PC circuits in the context of the HVZK proof systems and we wrote all 2PC circuits in the *mpc*-specific multi-party computation language (MPCL). We open-source our secure computation circuits here[3].

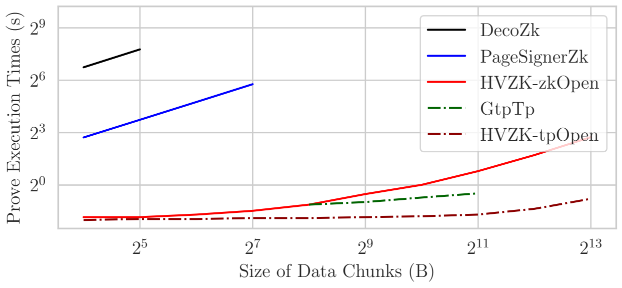


Fig. 10. Prove computations benchmarks of the client challenge. Solid and dotted lines indicate prove computations where TLS data privacy holds. Solid or dotted lines originate from proof systems with a transparent or trusted setup assumption. Dash-dotted lines indicate prove computations in which *clients* disclose TLS data. Lines closer to the bottom right corner are better and prove more data in less time.

B. 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).

1) Optimized Proof Computations for TLS 1.2 and TLS 1.3: To gain a fair comparison between related works [4], [6] and the optimization introduced in this work, we re-implemented privacy-preserving client challenges of related works with our software stack (cf. Section VI-A). For Deco [6], the zkSNARK circuit to prove private data (DecoZk) depends on the AES128 computation in the GCM mode. Our results show that DecoZk scales linearly with respect to the size of the opened plaintext data (cf. Figure 10). For a comparable setup with the same security assumptions, we evaluate the ZKP proofs on DecoZk using the *plonkFRI* proof system. This way, DecoZk does not rely on the assumption of a trusted setup. To recap, we manage to deploy the HVZK proof system in the asymmetric privacy setting, where the HVZK proof system and the unilateral validation do not require a trusted setup. We open source our AES128 GCM circuit implementation[4] as the first AES implementation in the ZKP framework *gnark*. Our AES implementation performs slightly worse compared to benchmarks found in the work [6]. This behavior is expected because our AES implementation uses a naive s-box lookup, which related implementations optimize [46]. Thus, compared to DecoZk configured with the *plonkFRI* proof system, our HVZK proof computation of the circuit $\mathcal{C}_{\mathbf{zkOpen}}$ is 382x faster at the data point where x-axis = 2^4 bytes (cf. Figure 10).

For the re-implementation of the zkSNARK circuit described by the work [5], we computed a zkSNARK friendly

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

[3]https://github.com/januspaper/submission1/tree/esp

[4]https://github.com/Consensys/gnark/pull/719

TABLE III
END-TO-END BENCHMARKS OF THE *Janus* PROTOCOL AND TLS 1.3 AS THE BASELINE. FOR THE LAN SETTING, WE ASSUME A ROUND-TRIP-TIME RTT=0 MS AND A TRANSMISSION RATE R_t=1 GBPS. VALUES MARKED WITH " TAKE ON THE VALUE OF THE PREVIOUS ROW.

	Communication (kb)				**Execution LAN (s)**			
Protocol	Offline	Handshake	Record	Post-record	Offline	Handshake	Record	Post-record
TLS 1.3	-	1.94	16.28	-	-	0.016	0.001	-
$Janus_{\text{transparent}}$	305.17 (MB)	113.8	984	583	1.99	0.51	1.04	0.46
$Janus_{\text{private}}$	406.29 (MB)	"	"	2 (MB)	2.63	"	"	2.08

commitment on the 2PC output wire labels of the circuit $\mathcal{C}_{\text{CB}_{2+}}$. With the *mpc* framework [44], where the circuit encryption function encrypts 128-bit wire keys, committing to 16 bytes of private TLS data requires a commitment on 2 kilo bytes of wire keys at the circuit output. With computational resources of the MacBook Pro, we were able to evaluate a zkSNARK-friendly MiMC commitment in the *gnark* framework on a maximum of 128 bytes (cf. PageSignerZk in Figure 10). Thus, compared to proof computations of private TLS data according to the Pagesigner protocol [5], our optimization achieves an improvement factor of 152x at the data point where x-axis $= 2^7$ bytes (cf. Figure 10).

Last, we compare an HVZK proof evaluation of the circuit $\mathcal{C}_{\textbf{tpOpen}}$ against the benchmarks presented in the work [15], which authenticates TLS data transparently. Compared to the garble-then-prove approach based on semi-honest 2PC using authenticated garbling [15], we verify transparent data presentations 2.3x faster at the data point where x-axis $= 2^{11}$ bytes.

2) Optimized End-to-end Performance: We present end-to-end performance benchmarks in Table III and evaluate a typical API traffic transcript which encompasses a 256 byte query and a 2 kB response. In the evaluation scenario, only client application traffic secrets are derived because in the case of a single response, the *client* can compute response CBs after obtaining the session secrets of the *proxy*. The end-to-end times build upon the micro benchmarks of 2PC subcircuits which are presented in the Appendix A. The TLS 1.3 baseline serves as a reference. Additionally, we measure 321 milliseconds for the integrated 3PHS and a proceeding ECTF conversion. The out-of-circuit verification of SHTS and the server certificate verification take 9.26 milliseconds in total. Pre-processing of TLS parameters for the client challenge takes 4.47 milliseconds.

The resulting end-to-end benchmarks show that the computation overhead of the *Janus* protocol mainly affects the record phase. We explain this trend with the semi-honest 2PC of the circuit $\mathcal{C}_t$ which depends on algebraic operations introduced by the polynomial over GF(128). Concerning private TLS data proofs, our post-record execution timings establish new standards for privacy-preserving TLS oracles. Our end-to-end benchmarks in the local area network (LAN) setting serve as a comparison baseline for future works.

VII. DISCUSSION

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

A. Related Works

The garble-then-prove paradigm of the work [15] introduces semi-honest 2PC based on authenticated garbling to improve efficiency of TLS oracles. In contrast, we show that in TLS 1.3 SHTS authenticity when the *verifier* and *client* securely evaluate the circuit $\mathcal{C}_{\text{SHTS}}$. With access to an authentic SHTS parameter, the *verifier* can check the correctness of semi-honest 2PC computations of the garble phase in the subsequent prove phase. Thus, our work is able to deploy semi-honest 2PC system for the computations in the garble phase. Additionally, we show how a HVZK proof system based on semi-honest 2PC can be made maliciously secure. To do so, we introduce a new unilateral validation phase for the *client*. The related work [15] derives a Pedersen commitment from the semi-honest 2PC based on authenticated garbling, which is supposed to be opened in zkSNARK proof systems. In contrast, we evaluate the client challenge in a HVZK system which efficiently evaluates non-algebraic algorithms.

The work [13] leverages the structure of AEAD stream ciphers and demands clients to commit to stream cipher CBs via a *pad commitment*. Concerning the commitment to AEAD stream ciphers CBs, the ZKP computation involves the computation of TLS legacy algorithms (e.g., AES128), which are of non-algebraic structure. The related work [13] notices that legacy algorithms contribute to over 40% of ZKP computation times and improves the efficiency of their protocol by putting the proof computation of the *pad commitment* into a pre-processing phase. Our work, on the contrary, directly computes the stream ciphers in the dedicated HVZK proof system based on boolean garbled circuits, which efficiently verify non-algebraic structures.

The works [4], [5] decouple the maliciously secure 2PC evaluation of response CBs through $\mathcal{C}_{\text{CB}_{2+}}$, where the *client* obtains output wire keys and shares a commitment of CB wire keys with the *verifier*. With the commitment, the *verifier* discloses the wire key decoding table as well as secret shares with the *client*. The *client* is now able to verify the correctness of $\mathcal{C}_{\text{CB}_{2+}}$, access response data, and select a transparent data opening. Optionally, *clients* can prove TLS data in a ZKP circuit which (i) takes in private output wire keys, (ii) computes CBs with the decoding table as public input, and (iii) authenticates TLS data by XORing a plaintext with CBs to the intercepted ciphertext. This approach has the following limitation. The wire key possession before obtaining a decoding table prevent the *client* from accessing response data such that the *client* remains with two options. With knowledge of the plaintext structure, the client commits to a selection of output encodings, which correspond to the CBs of interest for the privacy-preserving data opening. Without knowledge of the plaintext structure, the client uses a merkle tree commitment structure to commit to all output encodings and selectively opens CBs in the ZKP circuit via merkle tree inclusion proofs [5]. Due to frequent updates, API data is unlikely to remain static over a longer period of time such that the scenario of not knowing plaintext structures prevails. And, the introduction of the merkle tree increases the complexity of privacy-preserving proofs, which scale with the amount of commitments. Our work, in contrast, allows clients to selectively prove plaintext data during the client challenge.

B. Limitations & Future Work

The current version of the *Janus* protocol is tailored to the conditions found in TLS 1.3. Researching how to derive authentic handshake parameters in a malicious setting for TLS 1.2 remains future work. Due to the fact that the related works [5], [15], [32] withhold benchmarks of privacy-preserving client challenges, we evaluate and compare proof computations against our re-implementations (cf. Figure 10). Another limitation is that our implementation computes record authentication tags using general-purpose 2PC systems even though the work [5] proposes an optimized alternative to compute authentication tags with secure 2PC. Further, recent enhancements of the Multiplicative to Additive (MtA) algorithm have been proposed [47] which improves the efficiency of ECTF.

We expect that the underlying paradigm of the *Janus* protocol can be similarly applied to TLS 1.2. To recap, with the paradigm of our protocol, we develop a secure mutual verification of a handshake secret that is authenticated by the *server*. Similar to how our work uses the authenticity of SHTS, the authenticity of a TLS 1.2 handshake secret could be leveraged to employ the garble-then-prove scheme that relies on semi-honest 2PC only. However, the following differences must be considered. Securely deriving a parameter to bind the *server* authenticity differs in the context of TLS 1.2 because the server-side handshake messages are not immediately derived and communicated. Additionally, TLS 1.2 can employ encryption protocols which count as *key-committing* (e.g. AES in the CBC-HMAC mode) [13]. This factor could be leveraged to establish a maliciously secure authentication of server parameters. Further, investigating TLS 1.2 cipher suites with stronger security guarantees with regard to optimizing HVZK computations is an interesting direction of future work.

VIII. Conclusion

In this work, 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 of TLS 1.2 and TLS 1.3 oracles, we show that a HVZK proof system can be deployed if the *client* performs a unilateral validation of the *verifier*. Further, for TLS 1.3 oracles, we show that the authenticity of the SHTS secret can be mutually verified in a malicious adversary setting. We leverage the authenticity guarantees of SHTS to replace the 2PC system in the garble-then-prove paradigm with a lightweight 2PC system which does not require authenticated garbling. Our contributions improve the efficiency of the client challenge and establish new end-to-end benchmarks for TLS 1.3 oracles.

IX. Acknowledgements

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] L. Rosenthol, "C2pa: the world's first industry standard for content provenance," in *Applications of Digital Image Processing XLV*, vol. 12226. SPIE, 2022, p. 122260P.

[2] S. Longpre, R. Mahari, A. Chen, N. Obeng-Marnu, D. Sileo, W. Brannon, N. Muennighoff, N. Khazam, J. Kabbara, K. Perisetla *et al.*, "The data provenance initiative: A large scale audit of dataset licensing & attribution in ai," *arXiv preprint arXiv:2310.16787*, 2023.

[3] H. Ritzdorf, K. Wüst, A. Gervais, G. Felley, and S. Capkun, "Tls-n: Non-repudiation over tls enabling-ubiquitous content signing for disintermediation," *Cryptology ePrint Archive*, 2017.

[4] "Tlsnotary–a mechanism for independently audited https sessions." https://github.com/tlsnotary/how_it_works/blob/master/how_it_works.md, 2014.

[5] "Pagesigner: One-click website auditing." https://old.tlsnotary.org/how_it_works, 2023.

[6] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels, "Deco: Liberating web data using decentralized oracles for tls," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1919–1938.

[7] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*, 2016, pp. 270–282.

[8] J. Ernstberger, J. Lauinger, F. Elsheimy, L. Zhou, S. Steinhorst, R. Canetti, A. Miller, A. Gervais, and D. Song, "Sok: Data sovereignty," *Cryptology ePrint Archive*, 2023.

[9] D. Malkhi. (2023) Exploring proof of solvency and liability verification systems. [Online]. Available: https://blog.chain.link/proof-of-solvency/

[10] J. Frankle, S. Park, D. Shaar, S. Goldwasser, and D. Weitzner, "Practical accountability of secret processes," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 657–674.

[11] K. Balan, S. Agarwal, S. Jenni, A. Parsons, A. Gilbert, and J. Collomosse, "Ekila: Synthetic media provenance and attribution for generative art," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 913–922.

[12] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, "Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 191–219.

[13] C. Zhang, Z. DeStefano, A. Arun, J. Bonneau, P. Grubbs, and M. Walfish, "Zombie: Middleboxes that don't snoop," *Cryptology ePrint Archive*, 2023.

[14] M. Jawurek, F. Kerschbaum, and C. Orlandi, "Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 955–966.

[15] X. Xie, K. Yang, X. Wang, and Y. Yu, "Lightweight authentication of web data via garble-then-prove," *Cryptology ePrint Archive*, 2023.

[16] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the tls 1.3 handshake protocol," *Journal of Cryptology*, vol. 34, no. 4, pp. 1–69, 2021.

[17] J. Len, P. Grubbs, and T. Ristenpart, "Partitioning oracle attacks," in *30th USENIX security symposium (USENIX Security 21)*, 2021, pp. 195–212.

[18] N. J. Al Fardan and K. G. Paterson, "Lucky thirteen: Breaking the tls and dtls record protocols," in *2013 IEEE symposium on security and privacy*. IEEE, 2013, pp. 526–540.

[19] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Lucky 13 strikes back," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 85–96.

[20] Y. Sheffer, R. Holz, and P. Saint-Andre, "Recommendations for secure use of transport layer security (tls) and datagram transport layer security (dtls)," Tech. Rep., 2015.

[21] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the tls 1.3 handshake protocol candidates," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 1197–1210.

[22] C. Adams, S. Farrell, T. Kause, and T. Mononen, "Internet x. 509 public key infrastructure certificate management protocol (cmp)," Tech. Rep., 2005.

[23] Y. Lindell, "Secure multiparty computation for privacy preserving data mining," in *Encyclopedia of Data Warehousing and Mining*. IGI global, 2005, pp. 1005–1009.

[24] A. C.-C. Yao, "How to generate and exchange secrets," in *27th annual symposium on foundations of computer science (Sfcs 1986)*. IEEE, 1986, pp. 162–167.

[25] Y. Huang, "Practical secure two-party computation," *dated: Aug*, 2012.

[26] T. Chou and C. Orlandi, "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, 2015, pp. 40–58.

[27] Y. Huang, J. Katz, and D. Evans, "Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 272–284.

[28] A. Nitulescu, "zk-snarks: a gentle introduction," 2020.

[29] J. Thaler *et al.*, "Proofs, arguments, and zero-knowledge," *Foundations and Trends® in Privacy and Security*, vol. 4, no. 2–4, pp. 117–660, 2022.

[30] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "Sok: General purpose compilers for secure multi-party computation," in *2019 IEEE symposium on security and privacy (SP)*. IEEE, 2019, pp. 1220–1237.

[31] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[32] S. Celi, A. Davidson, H. Haddadi, G. Pestana, and J. Rowell, "Distefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more," *Cryptology ePrint Archive*, 2023.

[33] R. Gennaro and S. Goldfeder, "Fast multiparty threshold ecdsa with fast trustless setup," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1179–1194.

[34] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 21–37.

[35] D. Demmler, G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, and S. Zeitouni, "Automated synthesis of optimized circuits for secure computation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1504–1517.

[36] C. Hazay, P. Scholl, and E. Soria-Vazquez, "Low cost constant round mpc combining bmr and oblivious transfer," *Journal of cryptology*, vol. 33, no. 4, pp. 1732–1786, 2020.

[37] P. Grubbs, J. Lu, and T. Ristenpart, "Message franking via committing authenticated encryption," in *Annual International Cryptology Conference*. Springer, 2017, pp. 66–97.

[38] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish, "{Zero-Knowledge} middleboxes," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4255–4272.

[39] S. Gueron, "Key committing aeads," *Cryptology ePrint Archive*, 2020.

[40] S. Menda, J. Len, P. Grubbs, and T. Ristenpart, "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, 2023, pp. 379–407.

[41] Y. Dodis, P. Grubbs, T. Ristenpart, and J. Woodage, "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, 2018, pp. 155–186.

[42] A. Albertini, T. Duong, S. Gueron, S. Kölbl, A. Luykx, and S. Schmieg, "How to abuse and fix authenticated encryption without key commitment," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3291–3308.

[43] D. Amyot, "Paillier cryptosystem implemented in Go," https://github.com/didiercrunch/paillier, 2023.

[44] M. Rossi, "Secure Multi-Party Computation (MPC) with Go," https://github.com/markkurossi/mpc, 2023.

[45] G. Botrel, T. Piellard, Y. E. Housni, I. Kubjas, and A. Tabaie, "Consensys/gnark: v0.8.0," Feb. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.5819104

[46] A. Kosba, C. Papamanthou, and E. Shi, "xjsnark: A framework for efficient verifiable computation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 944–961.

[47] H. Xue, M. H. Au, M. Liu, K. Y. Chan, H. Cui, X. Xie, T. H. Yuen, and C. Zhang, "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*, 2023, pp. 2974–2988.

[48] P. Paillier, "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, 1999, pp. 223–238.

[49] C. Reitwiessner, "zksnarks in a nutshell," *Ethereum blog*, vol. 6, pp. 1–15, 2016.

[50] A. R. Block, A. Garreta, J. Katz, J. Thaler, P. R. Tiwari, and M. Zajac, "Fiat-shamir security of fri and related snarks," *Cryptology ePrint Archive*, 2023.

[51] J. Lu and J. Kim, "Attacking 44 rounds of the shacal-2 block cipher using related-key rectangle cryptanalysis," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 91, no. 9, pp. 2588–2596, 2008.

Appendix

A. Extended Performance Evaluation

The following subsection provides additional micro benchmarks.

1) Benchmarking 2PC Circuits: We present micro benchmarks of secure computation building blocks in Table IV. The table 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 $\mathcal{C}_{\text{XHTS}}$ and $\mathcal{C}_{\text{k,iv}}$ derive session secrets in milliseconds and compute CBs via the circuit $\mathcal{C}^{\text{X}}_{\text{CB}_{2+}}$ for a 2 kB record in 164.9 milliseconds. An interesting fact to notice is that the AEAD tag circuit $\mathcal{C}_{tag}$ is efficient for small request sizes and scales sufficiently but not ideally for larger request sizes. The overhead in the circuit $\mathcal{C}_{tag}$ 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 [5], [32] propose a scalable OT-based computation of the AEAD tag, which we consider as future work to improve our implementation.

Concerning data opening times, we can see that the *transparent* mode with the circuit $\mathcal{C}_{\text{tpOpen}}$ is more efficient compared to the *privacy-preserving* mode with the circuit $\mathcal{C}_{\text{zkOpen}}$. This behavior is expected because, the 2PC circuit of the transparent mode does not include the ciphertext, SHTS, and CB$_{tag}$

TABLE IV
SECURE COMPUTATION BENCHMARKS SEPARATED INTO OFFLINE/ONLINE EXECUTION AND COMMUNICATION VALUES. WE SEPARATE THE HANDSHAKE, RECORD, AND POST-RECORD PHASES WITH DASHED LINES.

2PC Circuit	**Constraints** (x10^6)	**Execution Offline**	**Execution Online**	**Communication Offline**	**Communication Online**
ECTF	-	-	212.96 ms	-	1.861 kB
$\mathcal{C}_{\text{XHTS}}$	3.14	215.56 ms	144 ms	34 MB	110 kB
$\mathcal{C}_{\text{k}^{m_1},\text{iv}}$	10.34	723.96 ms	484.82 ms	108.08 MB	356 kB
$\mathcal{C}^{256\text{ B}}_{\text{ECB}_{2+}}$ / $\mathcal{C}^{2\text{ kB}}_{\text{ECB}_{2+}}$	1.16 / 9.18	67.78 / 578.76 ms	67.6 / 164.9 ms	10.12 / 86.02 MB	116 / 566 kB
$\mathcal{C}^{256\text{ B}}_{tag}$ / $\mathcal{C}^{2\text{ kB}}_{tag}$	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
$\mathcal{C}^{256\text{ B q, 2 kB r}}_{\text{tpOpen}}$	12.69	0.89 s	0.46 s	126.01 MB	583 kB
$\mathcal{C}^{256\text{ B q, 2 kB r}}_{\text{zkOpen}}$ / 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

verification inside the circuit (cf. Figure 9). 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 IV), we evaluate the verification of a confidential document hash $H(f)$ in the circuit $\mathcal{C}_{\text{zkOpen}}$. To do so, we set the function f_ϕ=$H(f) \stackrel{?}{=} H(\mathbf{pt})$ 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.

B. 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.

1) Three-party Handshake: In the 3PHS (cf. Figure 11), 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 11, we show a TLS 1.3-specific configuration based on the ECDHE, where the parameters (e.g. Z_p) are EC points structured as $P=(x,y)$.

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.

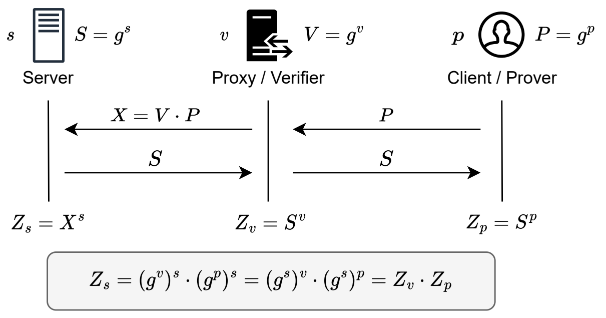


Fig. 11. Illustration of the 3PHS and exchanged cryptographic parameters between the server, the proxy, and the client. The gray box at the bottom indicates the relationship between shared client-side secrets Z_v and Z_p, which corresponds to the session secret Z_s of the server.

- **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.

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}.
- **hkdf.exp**$(k_{\text{pr}}, s_{\text{info}}, l) \rightarrow (k_{\text{okm}})$ 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 2), 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).

$$\begin{aligned}\textbf{hmac}(k,m) =& H((k' \oplus opad)||H((k' \oplus ipad)||m)) \\ & \text{with } k' = H(k), \text{ if } len(k) > b \\ & \text{and } k' = k, \text{ else}\end{aligned} \tag{2}$$

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) $\rightarrow$ (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) $\rightarrow$ (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) $\rightarrow$ $\{pt, \varnothing\}$ 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 $\varnothing$.

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 [23]. 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) 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) [48]. 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 [33], with the following tuple of algorithms, where

- **mta.Setup**(1^λ) $\rightarrow$ (sk_P,pk_P) takes in the security parameter λ and outputs a Paillier key pair (sk_P,pk_P).
- **mta.Enc**($\mathbf{x}$,sk_P) $\rightarrow$ ($\mathbf{c1}$) takes in a vector of field elements $\mathbf{x}$=$[x_1, \ldots, x_l]$ and a private key sk_P and outputs a vector of ciphertexts $\mathbf{c1}$=$[E_{sk_P}(x_1), \ldots, E_{sk_P}(x_l)]$.
- **mta.Eval**($\mathbf{c1}$,$\mathbf{y}$,pk_P) $\rightarrow$ (c_2,β) takes in the vector of ciphertexts $\mathbf{c1}$=$[\text{c1}_1, \ldots, \text{c1}_l]$, a vector of field elements $\mathbf{y}$=$[y_1, \ldots, y_l]$, and a public key pk_P. The output is a tuple of a ciphertext $\text{c2} = \text{c1}_1^{y_1} \cdot \ldots \cdot \text{c1}_l^{y_l} \cdot E_{pk_P}(\beta')$ and the share $\beta = -\beta'$, where $\beta' \xleftarrow{\$} \mathbb{Z}_p$.
- **mta.Dec**(c2,sk_P) $\rightarrow$ (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 $\mathbf{c1}$ 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 $\mathbf{x}$ and obtains the additive share α. Party p_2 inputs the private vector of multiplicative shares $\mathbf{y}$ 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.

b) ECTF Conversion: The ECTF algorithm is a secure 2PC protocol and converts multiplicative shares of two EC x-coordinates into additive shares [4], [6]. Figure 12 shows the computation sequence of the ECTF protocol which makes use the vector MtA algorithm defined in Section B5a. 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 [4], [6]. 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.

c) 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 [26]. 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) $\rightarrow$ (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}) $\rightarrow$ (X) takes in pp_{OT}, samples $x \xleftarrow{\$} \mathbb{Z}_p$, and outputs an encrypted secret $X = E_1(x)$.
- **ot.TransferY**(pp_{OT}, X, b) $\rightarrow$ (Y, k_D) takes in pp_{OT}, a cipher X, a bit b, and samples $y \xleftarrow{\$} \mathbb{Z}_p$. 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 $b \overset{?}{=} 1$.

```
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 $p_2$
$p_1$: $\rho_1 \xleftarrow{\$} \mathbb{Z}_p$; c1=mta.Enc($[-x_1,\rho_1]$, $sk$);
    send c1 to $p_2$
$p_2$: $\rho_2 \xleftarrow{\$} \mathbb{Z}_p$;(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,$sk$); $s_1 = 2 \cdot \alpha + \lambda_1^2 - x_1$
```

Fig. 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 $P_1 + P_2$.

- **ot.Encrypt**(pp_{OT}, X, Y, m_1, m_2, x) $\rightarrow$ (**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) $\rightarrow$ (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**.

d) Semi-honest 2PC with Garbled Circuits: We define secure 2PC based on boolean garbled circuits by extending our OT definition of Section B5c with the tuple of algorithms, where

- **gc.Setup**(1^λ) $\rightarrow$ (pp_{GC}) takes in the security parameter λ and outputs public parameters pp_{GC}.
- **gc.Garble**(pp_{GC}, $\mathcal{C}_G$, d_{in}) $\rightarrow$ ($\mathbf{k}_{in}^g$, **e**, $\boldsymbol{G}(\mathcal{C})$, $T_{\text{k-d}}$, $T_{\text{d-k}}$) takes as input pp_{GC}, a boolean circuit $\mathcal{C}_G$, the input bit string d_{in}, and randomly samples signal bits and wire keys σ,$k \xleftarrow{\$} \mathbb{Z}_n$. 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 $\boldsymbol{G}(\mathcal{C})$, 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**, $\boldsymbol{G}(\mathcal{C})$) $\rightarrow$ (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**, $\mathbf{k}_{in}^g$, $\boldsymbol{G}(\mathcal{C})$, 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 B5c. 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}.

e) Maliciously Secure TwoPC based on dual-execution: We consider running the semi-honest 2PC protocol based on boolean garbled circuits [24] 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.

6) Zero-knowledge Proof Systems: In practice, zero-knowledge proof systems are implemented by a tuple of algorithms, where

- **zk.Setup**(1^λ, $\mathcal{C}$) $\rightarrow$ ($\text{CRS}_\mathcal{C}$) takes in a security parameter and algorithm, and yields a common reference string,
- **zk.Prove**($\text{CRS}_\mathcal{C}$, x, w) $\rightarrow$ (π) consumes the CRS, public input x, and the private witness w and outputs a proof π.
- **zk.Verify**($\text{CRS}_\mathcal{C}$, x, π) $\rightarrow \{0, 1\}$ 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) 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 [49]. 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 [50]), 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.

C. 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 III) and uses our formalized cryptographic building blocks (cf. Sections II-C, Appendix 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 [14] 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 $[r_1, r_2]= \Pi_{ss}$.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 $\mathcal{C}_{open}$ 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($\mathcal{C}_{open}$)*
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 $\mathcal{C}_{open}$ 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 $\mathcal{C}_{\text{open}}$. However, if $\mathcal{A}$ receives c upon disclosure of $\{\text{p} \setminus T_{k-d}\}$, 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 $\Pi_{\text{comp}}(\Pi_{sh2PC}$=arg1, $\mathcal{C}_{\text{open}}$=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.

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 a specific TLS 1.3 operation 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.

3) Construction 3: The third construction reduces the security requirements of cryptographic constructions in the garble-then-prove paradigm [15]. 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 C).

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 $\mathcal{C}_{open}$ implementing Π_{com}.Open*
- *a 2PC circuit $\mathcal{C}_{kdc+record}$ implementing the TLS 1.3 specification*
- *a 2PC circuit $\mathcal{C}_\phi$ 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($\mathcal{C}_{kdc+record}$)*
2) *$\Pi_{g\text{-}t\text{-}p}$.Garble: p_2 calls Π^1_{sh2PC}.Evaluate($\mathcal{C}_{kdc+record}$)*
3) *$\Pi_{g\text{-}t\text{-}p}$.Prove: Π_{comp}(Π^2_{sh2PC} , ($\mathcal{C}_{kdc+record}$ + $\mathcal{C}_{open}$ + $\mathcal{C}_\phi$) , Π_{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.

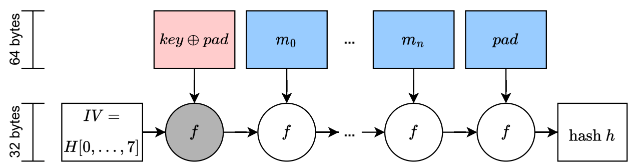


Fig. 13. Merkle-Damgård structure of the SHA256 hash function. Values marked in red indicate private input whereas blue background indicates public input. To protect a secret *key*, the prover must compute the first f (grey background) in-circuit. All remaining intermediate hash values f (white background) can be computed out-of-circuit by the verifier and checked against the public input hash value h.

D. HVZK Circuit Optimization

The TLS key derivation can be optimized by leveraging security guarantees provided by the Merkle-Damgård structure. The Merkle-Damgård structure is used to compress input data to a fixed size output in hash algorithms (e.g. SHA256). If TLS is configured with the cipher suite `TLS_AES_128_GCM_SHA256`, then the Merkle-Damgård repetitively appears during the TLS key derivation function **hkdf.extr** and **hkdf.exp** because the key derivation functions internally call **hmac**. The **hmac**algorithm calls SHA256 and with that the Merkle-Damgård structure.

1) Merkle-Damgård Structure: In a scenario where **hmac** is called in TLS 1.3 and TLS 1.3 is configured with `TLS_AES_128_GCM_SHA256`, the concatenation of the inner hash $H((K' \oplus ipad)||m)$ (32 bytes) and $K' \oplus opad$ (64 bytes) yields a 96 byte output, which in turn, is the input to the outer hash function (cf. Formula 2). The input to the inner hash function is of size $64 + \text{len}(m)$ bytes. Thus, both hash input sizes in HMAC are above 64 bytes. If the hash input of SHA256 is above 64 bytes, SHA256 applies the Merkle-Damgård structure which repeats calls to an internal compression blockcipher f to reduce the input to a fixed sized output. The compressing blockcipher SHACAL-2 of SHA256 uses 64 computation rounds to hide its input and has not been broken [51]. Thus depending on whether the inner or outer hash is computed, the first call of the one-way compression blockcipher inside SHA256 already hides inputs $(K \oplus ipad)$ or $(K \oplus opad)$ of size 64 bytes and with that, hides the secret K of the prover [6]. As a result, the output of the compressing blockcipher in SHA256 can be used as public input to reduce the HVZK circuit complexity.

Figure 13 shows the case which applies in a ZKP circuit to compute the HMAC inner hash $H_{inner} = H((K' \oplus ipad)||m)$, where e.g. $m = \mathbf{H}_2$ is publicly known input. If m is publicly known by the verifier, the prover can compute the grey f and disclose it to the verifier, which computes the remaining part of the hash out of circuit. The same optimization of SHA256 is feasible when computing the outer hash $H_{\text{outer}} = H((K' \oplus opad)||\mathbf{H}_{\text{inner}})$. Thus, proving HMAC in a ZKP takes two evaluations of the SHACAL-2 compression function f if the message input m is publicly known.

In the key derivation in TLS 1.3, successive SHA256 calls generate public intermediate values which allow intermediate proceedings of the TLS specification out of circuit. Generated intermediate values, which have been computed out of circuit, can be fed back into the HVZK circuit if the computation of secret parameters proceeds. Thus, all intermediate values which can be generated, computed on out of circuit, and fed back into the circuit as public input, optimize the HVZK circuit.