Janus: Fast Privacy-preserving Data Provenance for TLS 1.3

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 the data to any third party by using TLS oracles. Beyond that, TLS oracles can confirm the provenance and policy compliance of private online data by using zero-knowledge-proof systems. In practice, privacy-preserving TLS oracles can efficiently verify private data up to 1 kB in size selectively, preventing the verification of sensitive documents larger than 1 kB. In this work, we introduce a new oracle protocol for TLS 1.3, which reaches new scales in selectively verifying the provenance of confidential data. We tailor the deployment of secure computation techniques to the conditions found in TLS 1.3 and verify private TLS data in a dedicated proof system that leverages the asymmetric privacy setting between the client parties of TLS oracles. Our results show that 8 kB of sensitive data can be verified in 6.7 seconds, outperforming related approaches by 8x. With that, we enable new boundaries to verify the web provenance of confidential documents.**

***Index Terms*—TLS Oracles, Data Provenance, Zero-knowledge Proofs, Secure Two-party Computation, TLS 1.3.**

I. Introduction

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 another party, then the third party cannot verify if the presented data is *authentic* and *correct*, and, thus, cannot verify the provenance of the data. In the eyes of the third party, TLS data is *authentic* if the data origin can be verified. TLS data is *correct* if the third party is able to verify the integrity of presented TLS data against unforgeable TLS session parameters. To save the third party from verifying the data provenance itself, current approaches either consider *servers* to attest to shared data via digital signatures [1], or employ TLS oracles [2]–[4]. Data attestation through *servers* is an efficient data provenance solution but requires server-side software changes. In contrast, TLS oracles are *legacy-compatible* and achieve data provenance without introducing server-side changes. TLS oracles further introduce a trusted *verifier* to take over the verification of TLS data *authenticity* and *correctness*. If the data validation succeeds, the trusted *verifier* certifies the TLS data of clients. With the certificate, clients are able to convince any third party of data provenance if the third party trusts the verifier.

We clarify the functionality of TLS oracles with the use case of enhanced electronic surveillance (cf. Figure 1). The use case involves a law enforcement agency which requests a warrant from a judge for a sealed case [5]. If the judge accepts the warrant request, the judge issues a court order to

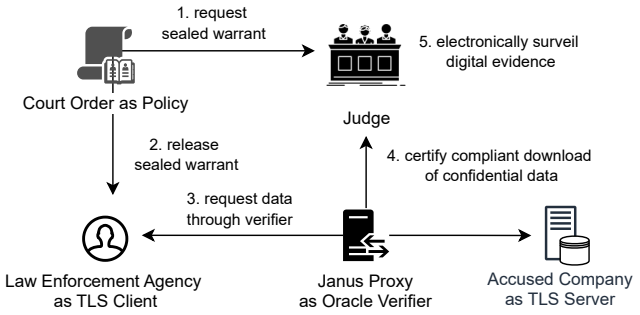


Fig. 1. Use case.

the law enforcement agency. The court order authorizes the law enforcement agency to collect confidential data from a targeted company. We suppose that for the data collection, the law enforcement agency queries a company database via TLS-protected API requests. Without a TLS oracle, the law enforcement agency can query files from the company but cannot convince the judge that the collected evidence has been obtained from the targeted company. By collecting data through a TLS oracle protocol, the law enforcement agency is able to prove to a judge that the evidence originated from the API endpoint of the company at a specific point in time. If the employed TLS oracle supports the verification of private data, then the operator of the TLS oracle learns nothing beyond the fact that the requested evidence complies with the requirements made in the court order.

This paper presents a TLS oracle protocol that we envision as helpful for an electronic surveillance use case. Law enforcement agencies depend on downloading confidential files and existing privacy-preserving TLS oracles are constrained in the amount of sensitive data they can validate. For instance, the work [4] presents an TLS oracle to selectively verify key value pairs of sensitive JavaScript Object Notation (JSON) data and validates 64 bytes of sensitive data per second (cf. Section VI). The work [3] verifies twice the amount of private data per second. However, the *client* is required to know the structure of queried data and cannot verify dedicated parts of TLS messages. In contrast, we introduce a new TLS oracle protocol for TLS 1.3 which achieves new scales in selectively verifying confidential data. Our approach validates 1 kB of sensitive data per second and outperforms related approaches by a factor of 8x. The applicability of our work goes beyond

the use case of verifying confidential documents and serves as a crucial technique to build user-centric and data-sovereign systems [6].

Before we outline the contributions of our work, we introduce preliminary technical details first. In TLS oracles, the client-side is a collaborative and secret-shared session between two parties: a trusted *verifier* acting as a *proxy* between the *client* and the *server*, and a *client*. In the presented use case of Figure 1, the *client* is represented by the law enforcement agency and the *verifier* is the operator of a proxy verification service. A secret-shared session splits TLS parameters into shares and guarantees that both the *proxy* and *client* locally maintain the shares. If the *proxy* and *client* interact with each other through secure computation techniques (e.g. secure two-party computation (2PC) [7]), shares can be merged together to reconstruct full TLS parameters. Secure computation techniques guarantee that the interaction to reconstruct parameters does not leak any information on individual shares. The secret-shared session introduces a mutual dependence between the *proxy* and the *client* such that both parties can only proceed according to the TLS specification if both parties interact with each other. This mutual dependence is used by the *proxy* to audit if the TLS data presented by the *client* originates from an authenticated and secret-shared TLS session. If the verification succeeds, the *proxy* attest the data provenance.

Our work improves the efficiency of verifying data provenance and builds upon two novel observations. The first observation concerns the key agreement protocol of the TLS 1.3 handshake phase. If the client hello (CH) message of the TLS key agreement protocol contains a key share that is supported by a cipher suite at the *server*, then the *server* immediately derives handshake traffic secrets and responds to the *client* with authenticated handshake messages. As a consequence, authenticated handshake messages can be expected before the computation of the server handshake traffic secret (SHTS). SHTS is used to verify server-side handshake messages, and, with that, verify the authenticity of the TLS session. In the context of TLS oracles, where the three-party handshake (3PHS) establishes secret-shared session parameters between the *proxy* and the *client*, the SHTS secret is computed in a secure computation protocol with security against malicious adversaries [2]–[4]. However, the same security guarantees hold if SHTS is computed in a more efficient secure computation protocol with security against semi-honest adversaries. When assuming semi-honest adversaries during the secure computation of SHTS, the *client* must verify SHTS against server-side handshake messages before disclosing SHTS to the *proxy* (cf. Section IV-B).

The second observation concerns the record and post-record phase of TLS oracles. The record phase exchanges requests and responses with the *server*. We notice that requests must be computed using maliciously secure computation protocols. Concerning the access to response data, the client-side requires secure computation only if the content of the response contributes to the compilation of a subsequent request of the same TLS session. Otherwise no secure computation is required due to our new post-record phase, where a mutual integrity verification between client parties yields an asymmetric privacy setting (cf. Section IV-C2). The post-record phase begins upon the *client*'s notification to the *proxy* that enough record data has been received. Up to this point, the *proxy* has recorded the traffic transcript of the TLS session and can audit *client* behavior against the intercepted transcript. However, the *proxy* is audited first and is required to disclose all locally maintained session secrets to the *client*. The disclosure of *proxy* session secrets initiates the asymmetric privacy setting, where only the *client* has access to all session secrets. With all session secrets, the *client* is able to detect a cheating *proxy* by locally recomputing and matching secure computation outputs against previously obtained outputs. Since the *client* is the only party with private secrets, the *proxy* can verify the *client* through a proof system which is based on semi-honest secure computation [8]. To do so, the *proxy* constructs the computation to be proven by the *client*. The computation of the proof system uses private inputs of the *client* to derive and match TLS session parameters against the recorded message transcripts of the *proxy*. To prevent the *client* from providing false inputs, we build upon our first contribution as follows. If the *proxy* honestly participates in the secure computation of SHTS after obtaining server-side handshake messages, then the *proxy* can securely verify the authenticity and correctness of SHTS. The proof system leverages the authenticity of SHTS to verify the correctness of *client* inputs by (i) deriving SHTS' from *client* inputs and by (ii) matching SHTS against SHTS'. Last, to prevent a malicious *proxy* from benefiting of the single bit leakage of the proof system [9], we introduce a commit-disclose-open paradigm (cf. Section IV-C3).

Our work achieves new efficiency gains compared to the related works [3], [4], [10]–[12] as follows. The computation of TLS parameters depends on non-algebraic structures which proof systems with boolean arithmetic efficiently verify. On the contrary, zkSNARK proof systems operate efficiently if algorithms leverage supported algebraic structures. Thus, the evaluation of AES128 on a 1 kB input takes 29 or 0.76 seconds in a *Groth16* zkSNARK or boolean-based proof system [8] respectively. With [8], our protocol verifies 8 kB of transparent or private web data in 0.58 or 6.7 seconds and sets new boundaries concerning the selective verification of data provenance. In analogy to Roman mythology, we name our efficient oracle solution after the god of transitions, Janus. Because, the *Janus* protocol guards the transition of large-scale web data into a representation where provenance can be verified. In summary,

- We introduce *Janus*, a novel protocol to verify the provenance of kilobytes of data, by tailoring secure computation techniques to the conditions found in TLS 1.3 while retaining security properties equivalent to previous works.
- We show that the honest behavior of the *proxy* leads to a guaranteed authenticity verification of the handshake secret SHTS and that the malicious behavior of the *proxy* does not yield any benefits in compromising the *client*.

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)))

- We improve the efficiency of selective and privacy-preserving data proofs by verifying the integrity of TLS data in a semi-honest zero-knowledge proof system, which leverages the asymmetric privacy setting between client parties (cf. Section V-C3).
- We analyse the security of *Janus* (cf. Appendix F1), provide end-to-end evaluation benchmarks (cf. Section VI), and open-source[1] the implementation of our secure computation building blocks.

II. PRELIMINARIES

This section introduces the key concepts of TLS 1.3 which oracle protocols modify or build upon such that the *authenticity* and *correctness* of TLS session data can be publicly verified. In addition, we explain the main functionalities of cryptographic building blocks and provide further details of each cryptographic construction or protocol in the Appendix A.

A. General Notations

The TLS notations of this work are introduced in Section II-B, and closely follow the notations of the work [13]. 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 $\cdot, \oplus$ represent the logical AND or multiplication, and the logical XOR or addition respectively. Other operators describe a random assignment of a variable with $\xleftarrow{\$}$, the concatenation of strings with $||$, and the comparison of variables with $\stackrel{?}{=}$.

B. Transport Layer Security

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

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

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_{CH}
2. s: $Y = y \cdot G$; send Y in m_{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: fk_S = **hkdf.exp**(SHTS, "finished" || "")
10. s: SCV=**ds.Sign**(sk_S,label_{11}||H_6); send SCV in m_{SCV}
11. s: SF = **hmac**(fk_S, H_7); send SF in m_{SF}
12. c: SF' = **hmac**(fk_S, H_7); verify SF' $\stackrel{?}{=}$ SF
13. c: **ds.Verify**(pk_S, label_{11} || H_6, SCV) $\stackrel{?}{=}$ 1
14. b: fk_C = **hkdf.exp**(CHTS, "finished" || "")
15. c: CF = **hmac**(fk_C, H_9); send CF in m_{CF}
16. s: CF' = **hmac**(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: To establish a secure channel between a server and a client, TLS 1.3 relies on the Diffie-Hellman key exchange (DHKE) to securely exchange cryptographic secrets between two parties (cf. Figure 2, lines 1-4). With TLS configured to use elliptic curve cryptography, parties protect secrets x, y with an encrypted representation X, Y. Next, the values X, Y are exchanged in plain via the CH and server hello (SH) messages m_{CH}, m_{SH}. With access to X, Y, both parties derive the Diffie–Hellman ephemeral (DHE) key, where DHE $= x \cdot y \cdot G = y \cdot X = x \cdot Y$ holds. Both parties use the DHE value to compute the handshake secret (HS), and, with that, derive the SHTS and client handshake traffic secret (CHTS) (cf. Figure 2, lines 5,6). SHTS and CHTS are used to derive client-side and server-side handshake keys and initialization vectors (cf. Figure 2, lines 7,8), which secure all

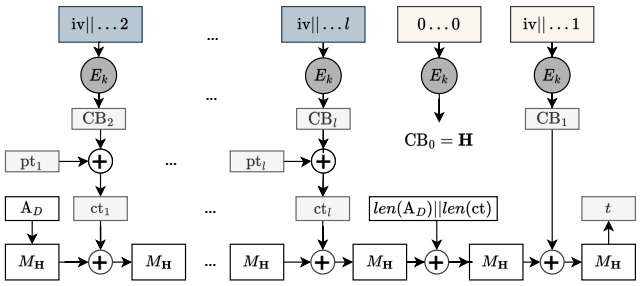
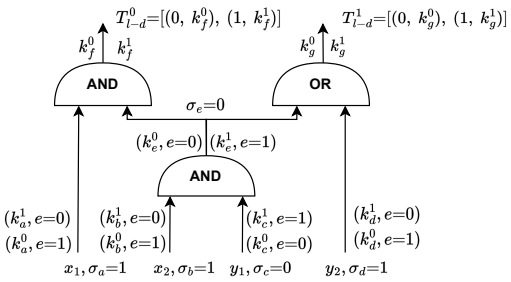


Fig. 3. AEAD stream cipher configured with AES in the Galois/Counter mode (GCM). The algorithm encrypts a plaintext $\mathbf{pt} = [pt_1, \ldots, pt_l]$ to a ciphertext $\mathbf{ct} = [ct_1, \ldots, ct_l]$ under key k and authenticates the ciphertext $\mathbf{ct}$ and associated data A_D with the tag t. The symbol $M_\mathbf{H}$ is a Galois field 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.



$G(\mathcal{C}^{0,(1,2)}_{\text{AND}})$	
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 which is computed using signal bits σ. The circuit $\mathcal{C}$ expresses the function f of a secure computation via boolean logic gates. Every circuit wire w_L is encoded with secret internal labels $\mathbf{i}$, 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 correspond to bits of input data. The lists $\boldsymbol{T}_{l-d}$ map output labels to output data. The gate-wise garbling tables $\boldsymbol{G}(\mathcal{C})$ map tuples of external labels to garbled labels concatenated with external labels. Output wires have neither external labels nor signal bits.

subsequent handshake messages.

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 server certificate verify (SCV) value, which binds a Public Key Infrastructure (PKI) X.509 certificate to the TLS 1.3 transcript via a digital signature [14]. 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.

The remainder of the TLS handshake phase computes the client application traffic secret (CATS) and server application traffic secret (SATS), which are used to compute server-side and client-side application traffic keys and initialization vectors (cf. Figure 2). The TLS record phase, which we describe next, continues to use the derived record parameters.

2) Record Phase: The TLS record phase requires parties to protect data with an AEAD algorithm before data can be exchanged. AEAD algorithms in TLS 1.3 use stream ciphers to protect data and depend on previously established application traffic keys and initialization vectors 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, which incrementally output key streams or counter blocks (CBs) (cf. Figure 3). In a next stage, 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 the AEAD algorithm in the Appendix D. Notice that only the parties of the TLS handshake phase have access to AEAD encryption and decryption secrets and are eligible to access exchanged record phase traffic.

C. Cryptographic Building Blocks

This section provides an overview of the cryptographic fundamentals that support the *Janus* protocol beyond algorithms found in TLS and related works. Formal descriptions of the cryptography and further details of oracle-specific algorithms can be found in the Appendix A.

1) Semi-honest 2PC with Garbled Circuits: 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 [7], [15]. A 2PC system based on boolean garbled circuits involves a party p_1 with input $\mathbf{x}$ as the garbler and party p_2 with input $\mathbf{y}$ 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 $\mathbf{i}$. 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 [16]. 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 circuit outputs, garbled entries encrypt wire keys only. For intermediate gates of the circuit, garbled entries encrypt wire keys concatenated with corresponding external labels.

After garbling the entire 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 Garbled Circuits (GCs) depends on the 1-out-of-2 OT_2^1 sub protocol to secretly exchange input parameters of the circuit [17]. 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 [17], 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 [18] to secure the semi-honest 2PC defined in Section II-C1 against malicious adversaries. The maliciously secure 2PC 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.

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 $R_{\mathcal{L}}(x, w) \stackrel{?}{=} 1$. The assumption is that for any statement $x \in \mathcal{L}$, there exist a valid witness w and no witness exists for statements $x \notin \mathcal{L}$. In order for proof systems to work, three properties of *completeness*, *soundness*, and *zero-knowledge* must hold. 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. Beyond that, the notion of honest verifier zero-knowledge (HVZK) holds if the zero-knowledge property can be shown for a semi-honest verifier, who honestly follows the protocol specification of the proof system [19], [20].

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 [8]. 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 of *completeness*, *soundness*, and HVZK of the HVZK proof system based on garbled circuits can be found in the work [8].

5) Cryptographic Commitments: A cryptographic commitment or simply commitment hides committed data in a commitment string that can only be computed based on an explicit and unequivocal mapping of input data. The computation of the commitment string based on valid input data is called the opening of the commitment. Commitments are characterized by two properties. The *binding* property prevents an adversary to find a second valid opening of the commitment. The *hiding* property guarantees that the commitment does not leak information of committed data.

III. System Model

The system model of the *Janus* protocol introduces system goals in form of security and usability properties and defines a threat model and system roles.

A. System Roles

Clients establish a TLS 1.3 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 secret shares of the *client*.

Servers participate in TLS 1.3 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.

Proxies take over the role of TLS oracle trusted verifiers. Proxies are configured at the client and route TLS traffic between the *client* and the *server*. We assume malicious *proxies* deviating from the protocol specification with the goal to learn TLS session secret shares of *clients*. Proxies honestly execute algorithms of the *Janus* protocol if the algorithm protects secret shares of *proxies*.

B. System Goals

The *Janus* protocol achieves system goals which we express via the following properties:

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 can neither modify server-side TLS traffic, no 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 full TLS 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. Depending on the operation mode of the *Janus* protocol, *session-confidentiality* is expected to break for *proxies* at some point in the protocol.

MITM-resistance guarantees that *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 and fresh randomness per TLS session between all interacting system roles. This means that the *proxy* cannot break TLS integrity and confidentiality. Network traffic, even if it is intercepted via a MITM attack by the *client*, cannot be blocked indefinitely. We assume up-to-date Domain Name System (DNS) records at the *proxy* such that *proxies* 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 *proxy*, 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. Depending on the employed ZKP systems, completeness, soundness, and HVZK hold.

IV. Protocol Overview

The focus of the protocol overview lies on briefly introducing existing cryptographic building blocks of related approaches, which the *Janus* protocol builds upon. Throughout this section, we clearly mark established building blocks and highlight novel observations made in this work (cf. Sections IV-B, IV-C).

A. The Three-party TLS Setting

TLS oracles turn the two-party protocol of TLS into a three-party protocol by introducing a trusted *verifier* [2]. The task of the newly introduced *verifier* is the verification and attestation of TLS data, which the *client* eventually presents. If the verification succeeds, the *verifier* attests to TLS data of the *client* by signing the data. The scope of data verification at the *verifier* ensures that (i) the presented data is authentic and originates from a TLS session between the *client* and an authenticated *server*, and (ii) the integrity of the presented TLS data holds, according to the TLS specification, via a verifiable computation trace.

1) Three-party Handshake: In order to audit the integrity of TLS data, the verifier and client join a mutually vetting but collaborative TLS client session. To construct a joint client-side session, TLS oracles replace the TLS handshake with a 3PHS [2], [4]. 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 A presents the cryptography of the 3PHS. Similarly to related approaches [2]–[4], the *Janus* protocol employs the 3PHS with the difference that the *verifier* acts as a *proxy* (cf. top of Figure 5).

The consequence of the 3PHS is that the *client* depends on the computational interaction with the *proxy* as a verifier to correctly proceed in the TLS protocol. At the same time, the *verifier* is convinced that the *client* preserves computational integrity according to the TLS specification if the joint TLS computation progresses. 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*. And, the introduction of false session data on the client-side leads to a session abort at the server. *Clients*, on the other hand, detect malicious behavior of the *verifier* during mutually dependent client-side computations.

2) Client-side Two-party Computation: With mutually dependent TLS secrets, the *client* and the *verifier* continue TLS computations by using secure two-party computation (2PC). Secure 2PC is a special type of interactive computation, which maintains input secrecy and, as such, secrecy of mutually injected secret shares. To achieve efficient secure 2PC, TLS oracles rely on different 2PC techniques. For instance, after the 3PHS, current approaches convert DHE secret shares in form of elliptic curve (EC) coordinates into bit-wise additive secret shares [2], [4]. Additive secret shares can be efficiently added together in 2PC circuits that are based on boolean GCs [7], [18], [21]. To convert elliptic curve secret shares into bit-wise additive secrets, current works use the Elliptic Curve to Field (ECTF) conversion algorithm [22]. In the same way as the works [2]–[4], the *Janus* protocol employs the ECTF algorithm (cf. Figure 5) and we present further details of the ECTF algorithm in the Appendix E2. After the ECTF conversion, the client-side continues to use maliciously secure 2PC based on boolean GCs to proceed according to the TLS-specification [2], [4], [10]. The reason for this is that the TLS key derivation and record layer encryption equally rely on algorithms, where GC-friendly and optimized binary circuits exist [23].

B. Handshake Secrets

1) SHTS: Similar to the work [4], we leverage the fact that, during the handshake phase, the *client* can securely disclose the SHTS parameter to the *proxy*. Even though the *proxy* knows SHTS, the key independence property of TLS 1.3 prevents the proxy from learning the HS secret and, with that, any application traffic keys which are derived from HS [24]. HS is protected by **hkdf.exp** through a pre-image resistant hash function (cf. line 5 and 17 of Figure 2). Our first contribution is that SHTS can be computed using semi-honest 2PC. However, a semi-honest 2PC computation of SHTS through a circuit $\mathcal{C}_{\text{SHTS}}$ only works for a specific case in TLS 1.3. If the *client* picks a *server*-supported cipher suite and a suitable key share in the CH message, then the *server* directly derives TLS session secrets and returns authenticated messages (cf. first three boxes in Figure 5). Notice that *clients* can query *servers* to discover supported cipher suites. If the *client* expects authenticated handshake messages right after sending the CH message, and verifies a semi-honest 2PC evaluation of $\mathcal{C}_{\text{SHTS}}$ against authenticated handshake messages, then no secret of the *client* can be extracted by the *proxy* (cf. Appendix G1).

If the *proxy* honestly garbles the 2PC circuit $\mathcal{C}_{\text{SHTS}}$ and verifies the respective SHTS output against authenticated messages of from the *server*, then authenticity for SHTS is guaranteed (cf. Appendix G4). Due to the fact that the *proxy* learns the transcript hash of the CH message first and since the *client* committed to a CH transcript hash with the *client* randomness,

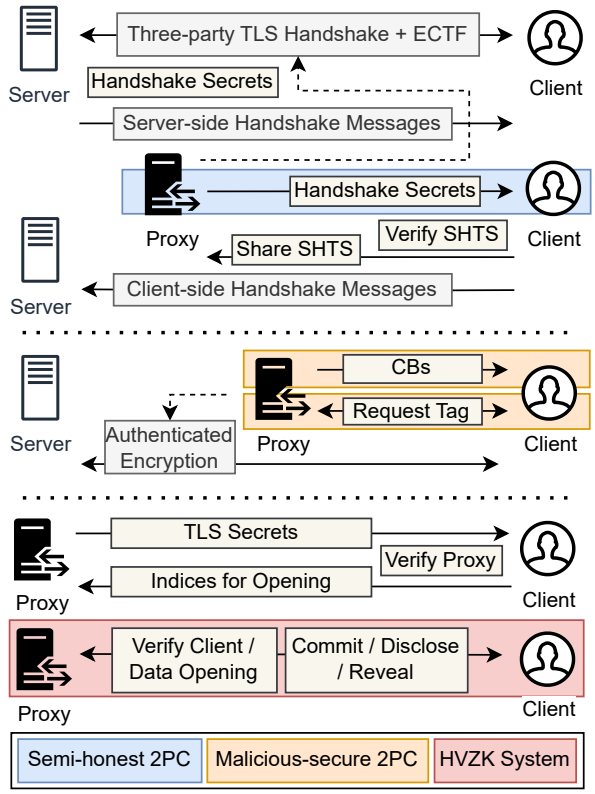


Fig. 5. Overview of the *Janus* protocol, where the *client* and a *proxy* collaboratively run a TLS session with the *server*. The dotted horizontal lines indicate the TLS handshake, TLS record, and post-processing phases. In the post-processing phase, the *client* continues to interact with the *proxy* in order to convince the *proxy* of a valid TLS data opening.

the *client* cannot replay authenticated handshake messages for another transcript that takes in the same CH message.

2) CHTS: Malicious garbling of the CHTS circuit does not yield any information leakage to the *proxy* either. Because, if an incorrectly garbled $\mathcal{C}_{\text{CHTS}}$ outputs the HS or secret share of the *client*, then the *client* computes client handshake traffic keys based on a CHTS' secret. If the *client* uses incorrect handshake traffic keys to compute client-side handshake messages in form of ciphertexts and authentication tags, then a *proxy* cannot learn any *client* secrets by viewing client-side messages (cf. Appendix G1).

C. TLS Records and Data Provenance

After the handshake phase, the *client* and *proxy* jointly compute requests to the *server* and process responses from the *server*. The message transcripts of the record phase are recorded at the *proxy*.

1) Record Computation: To compute a request, the *client* evaluates a maliciously secure 2PC circuit $\mathcal{C}_{CB}$ which yields CBs of the encryption algorithm to the *client* only. The *client* locally computes the ciphertext representation of the request by XORing the CBs with plaintext chunks. To finalize the request, the authentication tag must be computed in a maliciously secure 2PC of the circuit $\mathcal{C}_{tag}$, because the authentication tag is made public as a part of the request

record. If the *proxy* maliciously garbles $\mathcal{C}_{tag}$ such that the output tag' is set to a *client* secret, and the *client* authenticates the request with tag', then the *proxy* can access *client* secrets by intercepting the request record and *session-confidentiality* is compromised. Additionally, the maliciously secure 2PC evaluation of $\mathcal{C}_{tag}$ prevents a malicious *client* from mounting a MITM attack. If *clients* cannot compute valid authentication tags individually, *clients* cannot compile a second valid request that passes the *server* AEAD verification. The maliciously secure computation of request CBs prevents the following case. If the *proxy* maliciously garbles $\mathcal{C}_{CB}$ to output known CBs and honestly garbles $\mathcal{C}_{tag}$, then the *client* cannot detect false CBs. With knowledge of CBs, the *proxy* is able to decrypt ciphertext chunks and break *session-confidentiality*.

2) Mutual Verification of Session-integrity: The last phase of TLS oracles decouples the client-side 2PC dependence and requires *clients* to prove TLS data provenance towards *proxies*. Before the *client* challenges the *proxy* to verify data provenance, the post-processing phase mutually verifies *session-integrity* of both client-side parties. As such, the *proxy* is required to reveal all TLS session secrets. TLS secrets of the *proxy* can be disclosed once the *proxy* has collected enough transcript data to verify the provenance of session data. Since the *proxy* acts as a verifying service, disclosure of *proxy* secrets does not compromise the security properties provided by TLS. The *client* uses all session secrets to locally recompute and compare TLS session parameters with previously evaluated 2PC outputs and aborts the protocol upon detecting misbehavior. Otherwise, the *client* shares indices indicating the record chunks the *client* intends to open.

To verify *session-integrity* of the *client*, the *client* is required to interact with the *proxy* in a special 2PC protocol. The protocol requires the *client* to compute a proof using the 2PC-based HVZK proof system of the work [8], which we also refer to as the semi-honest proof system. The deployment of the HVZK proof system is feasible because (i) the 2PC-based proof system requires only the *client* to input private data and (ii) the *proxy* is incentivized to honestly verify the correctness of the *client*. We combine a commit-disclose-open paradigm (cf. Section IV-C3) with the semi-honest proof system, which introduces an additional commitment scheme and prevents the single bit leakage problem of the HVZK proof system [8], [9]. In order to compute the proof, the *client* evaluates derived session parameters, according to a circuit $\mathcal{C}_{zkOpen}$, against message transcripts which have been intercepted at the *proxy* (cf. Section V-C2). The *proxy* is only able to verify 2PC input correctness within $\mathcal{C}_{zkOpen}$ if the *proxy* has honestly garbled $\mathcal{C}_{\text{SHTS}}$ and successfully verified SHTS. If the $\mathcal{C}_{zkOpen}$ verification succeeds, then the *proxy* attests to the TLS data of the *client* by signing the data. With the signature of the *proxy*, data provenance of *client* data can be verified by any third party that trusts the *proxy*.

3) Commit-disclose-open Paradigm: The additional commitment scheme on 2PC output wire keys is important as leaking a single bit in semi-honest 2PC interaction could disclose private plaintext bits with high entropy. The commitment prevents the *proxy* as a malicious garbler to obtain any *client* secrets and gives the *client* the opportunity to verify garbling correctness of the 2PC circuit before the *client* shares wire keys with the *proxy*. The commit-and-reveal protocol works as follows. The *client* is required to share a commitment with randomness r_c of the HVZK output wire key with the *proxy*. Next the *proxy* discloses the garbling truth table to the *client* such that the *client* verifies honest garbling of the *proxy*. In the case of a malicious garbled truth table, the *client* aborts the protocol. Otherwise, the *client* shares the commit randomness r_c with the *proxy*. The *proxy* uses r_c to verify if the *client* committed to the correct outgoing wire keys and aborts the protocol if the verification fails.

V. Protocol Specification

The following sections provide the remaining details of the *Janus* protocol. The subsection V-B summarizes all required 2PC circuits. Afterwards, client-side parties mutually verify each other (cf. Section V-C), before the *proxy* attests to TLS data via two possible data attestation modes (cf. Section V-D).

A. System Setup

Our protocol builds upon system roles introduced in the Section III-A, where the *proxy* takes the role of the TLS oracle verifier. As such, the *proxy* acts as the garbler of all 2PC circuits and circuits for the computation of HVZK proofs. The *client* acts as the evaluator of all 2PC circuits and garbles 2PC circuits if a maliciously secure 2PC system is required. Initially, the *client* and *proxy* agree on a policy P, which indicates a public statement ϕ that holds on TLS data which is presented by the *client*. During the system setup, the *proxy* and *client* execute 2PC offline computations to set up all required parameters. Offline computations are independent of TLS session parameters (e.g., setup or preprocessing functions) and can be executed before a TLS session starts. Online computations are functions which must be called throughout the TLS session. With all offline parameters set, the *client* instantiates a TLS session with the *server* based on the 3PHS (cf. Appendix A). Subsequently, the *client* and *proxy* translate shared EC secrets of the DHE value into additive secret shares s_1, s_2 through the ECTF protocol (cf. Appendix E2). In the end, the *proxy* and *client* locally keep s_1 and s_2 respectively and it holds that $s_1 + s_2 = \text{DHE}$.

B. Two-party Computation

The *client* and *proxy* continue to interact with 2PC interactions to collaboratively but vetting follow the TLS specification. To do so, both parties input their secret shares s_1, s_2 into a number of 2PC circuits (cf. Figure 6), which disclose 2PC outputs to the *client*.

1) Handshake Circuits: The circuit $\mathcal{C}_{\text{XHTS}}$ is required to complete the handshake phase and, depending on the *client* or *server* input labels, computes the CHTS or SHTS secrets. The circuit takes as input both secret shares and initially derives $\text{DHE} = s_1 \oplus s_2$. If the circuit $\mathcal{C}_{\text{XHTS}}$ outputs SHTS to the *client*, then the *client* verifies the SF message. Upon

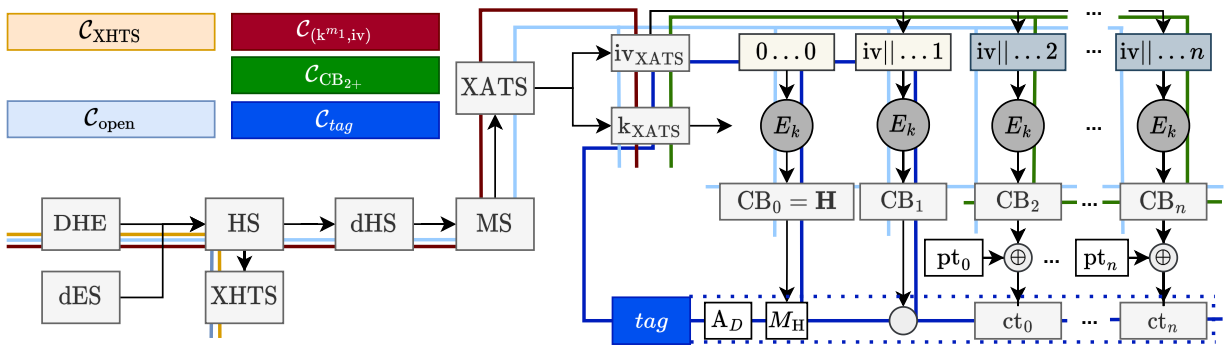


Fig. 6. Overview of 2PC circuits of the *Janus* protocol. All circuits are executed in a semi-honest adversary setting except the circuit $\mathcal{C}_{tag}$, which executes with security against malicious adversaries. Depending on provided inputs, the circuits output server-side or client-side TLS 1.3 parameters.

successful verification of the server certificate (SC), SCV, and SF messages, the *client* discloses SHTS to the *proxy*. If the circuit $\mathcal{C}_{\text{XHTS}}$ outputs CHTS, then the *client*, computes and sends the CF message to the *server* to finish the TLS handshake agreement. CHTS is never shared with the *proxy*.

2) Record Request Circuits: The next 2PC circuit $\mathcal{C}_{(\text{k}^{m_1},\text{iv})}$ requires the *proxy* to generate and input a random mask m_1, and takes as input both client-side secret shares. The circuit first reconstructs DHE $= s_1 \oplus s_2$ and, with DHE, derives application traffic secrets. The *client* obtains as output a masked application traffic key $k^{m_1}_{\text{XATS}} = m_1 \oplus k_{\text{XATS}}$ and the request initialization vector iv_{XATS}. Again, depending on the type of TLS 1.3 labels, the circuit outputs client or server application traffic keys.

Subsequently, for every request, the *proxy* and *client* compute the circuit $\mathcal{C}_{\text{CB}_{2+}}$, which outputs counter blocks $\mathbf{CB}_{2+}$ to the *client*. The notation $\mathbf{CB}_{2+}$ expresses counter blocks CB_i with an index $i > 1$. With access to $\mathbf{CB}_{2+}$ the *client* computes the ciphertext chunks $\mathbf{q}$ of the request record and shares $\mathbf{q}$ with the *proxy*. Once the *proxy* receives $\mathbf{q}$, both the *proxy* and *client* compute the 2PC circuit $\mathcal{C}_{tag}$ (cf. dark blue circuit in Figure 6) to complete the construction of the request. The circuit $\mathcal{C}_{tag}$ outputs the counter blocks $\mathbf{CB}_{tag}$=[CB_0, CB_1], which, together with the ciphertext chucks $\mathbf{q}$, let the *client* compute the authentication tag t locally. We follow the bandwidth-efficient 2PC dual-execution paradigm of the works [3], [18] to implement a malicious secure 2PC evaluation of the circuits $\mathcal{C}_{tag}$, $\mathcal{C}_{\text{CB}_{2+}}$, and $\mathcal{C}_{(\text{k}^{m_1},\text{iv})}$.

3) Record Response Circuits: After computing and sending requests to the *server*, the *proxy* records (i) request ciphertext and authentication tag pairs ($\mathbf{q}$, t^{q}) and (ii) proceeding response ciphertext and authentication tag pairs ($\mathbf{r}$, t^{r}) from the *server*. When processing server-side response data, the execution of the 2PC circuit $\mathcal{C}_{\text{ECB}_{2+}}$ depends on server application traffic secrets. Further, the computation of response CBs is only necessary if there exists a dependency between follow up requests and obtained responses:

Requests are independent of responses. If no request depends on the contents of a response, then the circuit $\mathcal{C}_{\text{ECB}_{2+}}$ is only called for the compilation of request ciphertexts. We show that response CBs can be locally computed by the *client* once *proxies* disclose session secrets in a later phase of the protocol (cf. Section V-C1).

Requests depend on responses. If a request with number $n > 1$ depends on the contents of responses $\mathbf{r}$=$[r_1, \ldots, r_l]$, where each response r_m has an index $m < n$, then the *client* and *proxy* 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 $\mathbf{r}$=$[r_1, \ldots, r_l]$ to construct the n-th request. Further, with access to the l-th $\mathbf{CB}_{2+}$, the *client* is incapable of accessing any TLS session secret share of the *proxy* and, thus, cannot compute any y-th $\mathbf{CB}_{2+}$, with $y>l$. To preserve *MITM-resistance*, it must hold that the *proxy* intercepts the pair ($\mathbf{r}, \text{t}^{\text{r}}$) before the circuit $\mathcal{C}_{\text{CB}_{2+}}$ outputs the corresponding $\mathbf{CB}_{2+}$ of response $\mathbf{r}$. Notice that no $\mathbf{CB}_{2+}$ vector includes any $\mathbf{CB}_{tag}$ parameters.

C. Post-record Verification

As soon as the *client* has exchanged enough record data with the *server* such that the statement ϕ can be satisfied, the *client* sends a notification to the *proxy* such that the *proxy* stops the recording of ciphertext authentication tag pairs. Next, the *Janus* protocol instantiates a mutual verification phase between the *proxy* and the *client*. During the mutual verification, the *client* verifies if the *proxy* (i) honestly garbled shared circuits and (ii) provided correct input data during 2PC interactions (cf. Section V-C1). The *proxy*, on the other hand, checks if the *client* preserves *session-integrity* and presents TLS parameters and data according to the TLS specification (cf. Section V-C3).

1) Proxy Verification: To initiate the proxy verification, the *proxy* discloses (i) locally maintained TLS session secrets and (ii) 2PC garbling truth tables to the *client*. The *client*, to preserve *session-confidentiality*, never discloses locally controlled secret shares. With full access to the session secrets, the *client* locally recomputes and matches all TLS session parameters against previously evaluated 2PC outputs. The *client* is convinced of an honest 2PC garbling by the *proxy* if the 2PC outputs match the individually computed TLS parameters.

Any detection of a malicious garbling at the *client* leads to a protocol abort. Subsequently, the *client* discloses recomputed request and response counter blocks $\mathbf{CB}_{tag}$ to the *proxy* such that the *proxy* is able to verify the authenticity of recorded traffic. A traffic pair ($\mathbf{x}$, $\mathbf{t}^{\mathrm{x}}$) is authentic if the counter blocks $\mathbf{CB}_{tag}$ and the ciphertext chunks $\mathbf{x}$ compute the corresponding tags $\mathbf{t}^{\mathrm{x}}$. After the disclose phase, the *client* is able to locally compute the remaining CBs of responses, which have not been decrypted up to this point. The local computation of response CBs is a strong efficiency improvement compared to the related works [2]–[4], [10], which perform the same computation as well as the response tag computation using maliciously secure 2PC.

2) Indices for Data Opening: With access to all record data in the clear, the *client* determines a list of indices $\mathbf{I}^{\mathrm{open}}$ and shares the list with the *proxy*. The list $\mathbf{I}^{\mathrm{open}}$ contains GCM indices of CB, ciphertext, and plaintext chunks. With $\mathbf{I}^{\mathrm{open}}$, the *proxy* can select required ciphertext chunks for the data provenance verification. Depending on the operation mode of the *Janus* protocol, the indices $\mathbf{I}^{\mathrm{open}}$ are used for a *transparent* or *privacy-preserving* verification of TLS 1.3 data.

Transparent Mode If the *Janus* protocol operates in the *transparent* mode, the *client* shares TLS plaintext chunks **pt** together with the list of indices $\mathbf{I}^{\mathrm{open}}$ with the *proxy*. The *proxy* verifies if the computation of presented plaintext chunks XORed with authentic CB chunks yields intercepted ciphertext chunks. The integrity verification of the *proxy* is based on a 2PC interaction, which ensures that the *client* knows session secrets that lead to an *authentic* derivation of CB chunks according to the TLS session (cf. Section V-C3).

Privacy-preserving Mode If the *Janus* protocol operates in the *privacy-preserving* mode, the *client* does not share TLS plaintext chunks, but, instead, shares $\mathbf{I}^{\mathrm{open}}$ and proves knowledge of authentic plaintext data, and evaluates the plaintext against the public statement ϕ (cf. Section V-C3). Again, the list of indices $\mathbf{I}^{\mathrm{open}}$ determines which plaintext chunks are (i) verified concerning *session-authenticity* and *session-integrity*, and (ii) validated against the statement ϕ.

3) Client Verification: The last 2PC interaction between the *proxy* and the *client* ensures that (i) the *client* inputs correct values throughout all 2PC interactions, (ii) the *client* releases correct 2PC outputs to the *proxy*, and (iii) the TLS 1.3 data presented by the *client* achieves *session-integrity* and *session-authenticity*. The 2PC interaction follows the HVZK proof system of the work [8], where a valid return of output wire keys convinces the *proxy* of a valid TLS data opening. The HVZK proof system is combined with the commit-disclose-open scheme (cf. Section IV-C3) to prevent any information leakage in the semi-honest proof system [9]. Depending on the *Janus* operation mode, the last 2PC interaction validates either a transparent data opening or keeps TLS data private such that only the validity of data compliance according to the statement ϕ is revealed to the *proxy*.

Transparent Data Opening In the transparent mode of *Janus*, the *proxy* garbles the semi-honest 2PC circuit $\mathcal{C}_{tpOpen}$, where only the *client* inputs private data in form of the shared secret s_2 (cf. Figure 7). The *proxy* reserves the disclosure of the decoding table $\boldsymbol{T}_{l-d}$ at this point such that the *client* obtains output wire keys only. The next phase is the commit-disclose-open scheme, where the *client* commits to the **tpOpen** outputs c_{tpo}=Com($k_{out}^{\mathrm{SHTS'}}||k_{out}^{\mathbf{CB'}}||k_{out}^{\mathbf{CB'}_{tag}}$, r) and returns the commitment c_{tpo} back to the *proxy*. After obtaining the commitment, the *proxy* shares the garbled truth tables $\boldsymbol{G}(\mathcal{C}_{tpOpen})$ including the output decoding tables $\boldsymbol{T}_{l-d}$ with the *client*. The *client* verifies correct garbling of the $\mathcal{C}_{tpOpen}$ circuit and shares the commitment randomness r with the *proxy* if the $\mathcal{C}_{tpOpen}$ circuit verification succeeds. The *proxy* uses the commitment randomness r to open the commitment c_{tpo} and is convinced of *session-integrity* and *session-authenticity* on presented TLS data if the following verification check succeeds: The *proxy* decodes committed wire keys and verifies if **CB'** together with the presented plaintext chunks lead to a matching computation of ciphertext chunks. The decoded SHTS' value must match the initially verified SHTS parameter and the $\mathbf{CB'}_{tag}$ parameter must match the previously shared $\mathbf{CB}_{tag}$ values (cf. Section V-C1).

Privacy-preserving Data Opening In the privacy-preserving mode, the *proxy* garbles the semi-honest 2PC circuit $\mathcal{C}_{zkOpen}$ (cf. Figure 7) and withholds the decoding and truth tables $\boldsymbol{T}_{l-d}$, $\boldsymbol{G}(\mathcal{C}_{zkOpen})$ from the *client* until the *proxy* receives the commitment c_{zko}=Com(k_{out}^{1}, r). Equally as in the transparent opening, the *client* proceeds according to the commit-disclose-open scheme and shares the commitment randomness r if the honest garbling verification succeeds. In the end, the *proxy* is convinced if the decoded output wire key of the commitment c_{zko} maps to a 1. The difference between the privacy-preserving and the transparent data opening is that the circuit $\mathcal{C}_{zkOpen}$ verifies **pt** and the derived **CB'** against **ct** as well as the parameters SHTS and $\mathbf{CB'}_{tag}$ in the circuit. As a result, **pt** remains hidden and the *proxy* only learns if or if not the private plaintext chunks comply with the statement ϕ. We express the statement compliance evaluation with the function call $f_{\phi}(\mathbf{pt})$ (cf. line 2 in Figure V-C2). To clarify

tpOpen(s_2 **;** s_1, $\mathbf{I}^{\mathrm{open}}$, **pt**):
1. SHTS', **CB'**, $\mathbf{CB'}_{tag}$ = $\mathcal{C}_{\mathrm{open}}(s_1, s_2, \mathbf{I}^{\mathrm{open}}, \mathbf{pt})$
2. return: $k_{out}^{\mathrm{SHTS'}}$, $k_{out}^{\mathbf{CB'}}$, $k_{out}^{\mathbf{CB'}_{tag}}$

zkOpen(s_2, **pt;** s_1, $\mathbf{I}^{\mathrm{open}}$, **ct=(q, r)**, $\mathbf{CB}_{tag}$, SHTS, ϕ):
1. SHTS', **CB'**, $\mathbf{CB'}_{tag}$ = $\mathcal{C}_{\mathrm{open}}(s_1, s_2, \mathbf{I}^{\mathrm{open}}, \mathbf{pt})$
2. **ct'** = **CB'** $\oplus$ **pt**
3. assert: SHTS'$\stackrel{?}{=}$SHTS; **ct'**$\stackrel{?}{=}$**ct**; $\mathbf{CB'}_{tag}\stackrel{?}{=}\mathbf{CB}_{tag}$
4. assert: $1\stackrel{?}{=}f_{\phi}(\mathbf{pt})$
5. return: k_{out}^{1}

Fig. 7. 2PC verification circuits for the transparent (**tpOpen**) or privacy-preserving (**zkOpen**) data opening. Both circuits are executed to verify if the *client* presents data which preserves *session-integrity* and *session-authenticity*. The semicolon **;** separates circuit inputs into private (left side) and public (right side) input arguments. The function f_{ϕ} validates input data against a public statement ϕ and returns a 1 if the validation succeeds.

TABLE II

SECURE COMPUTATION BENCHMARKS SEPARATED INTO OFFLINE/ONLINE EXECUTION AND COMMUNICATION VALUES. WE MARK MALICIOUSLY SECURE PROTOCOLS WITH $*$ AND 2PC PROOF SYSTEMS WITH Σ, AND 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
Σ $\mathcal{C}^{256\text{ B q, 2 kB r}}_{\text{tpOpen}}$	12.69	0.89 s	0.46 s	126.01 MB	583 kB
Σ $\mathcal{C}^{256\text{ B q, 2 kB r}}_{\text{zkOpen}}$ / 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

details of introduced parameters, we provide an example of a private data opening next.

Example: We assume that TLS is configured to use AES as the encryption function. If proving TLS data according to the statement ϕ depends on the data inside the third ciphertext chunk of the first response (r_1, t^{r}_1), then the index=3 is included in the list of indices $\mathbf{I}^{\text{open}}$. 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. $ct'_5 \stackrel{?}{=} ct_5$), 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*. Thus, pt_5 can be further evaluated against ϕ.

D. Data Attestation

Once the *client* verification at the *proxy* succeeds, the *proxy* attest to presented *client* data. The *Janus* protocol supports two attestation modes. In the transparent mode, the *proxy* 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 *proxy* overwrites the statement $\phi = H(\mathbf{pt})$ to the hash of verified data such that every third party can evaluate arbitrary statements against the attested and public data. With p_{cert}, the *client* gains public verifiability of TLS 1.3 data and can present p_{cert} to any third party who trusts the *proxy*. In the privacy-preserving mode, the structure of p_{cert} remains the same except that the statement ϕ expresses the compliance constraints as a string. By verifying a privacy-preserving attestation, any third party trusting the *proxy* learns that the *client* successfully verified TLS 1.3 data against the statement ϕ at time t.

VI. PERFORMANCE EVALUATION

The evaluation comprises a description of the software stack, provides benchmarks on a circuit level, compares the protocol scalability, and evaluates end-to-end metrics.

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 TLS 1.3 with the cipher suite *TLS_AES_128_GCM_SHA256* and we implement the ECTF and Multiplicative to Additive (MtA) conversion algorithms in Golang by using the Paillier cryptosystem [25]. For a coherent implementation of the *Janus* protocol in Golang, we chose the *mpc* library [26] to access secure 2PC based on garbled circuits and we chose the *gnark* framework [27] to re-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 2PC 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].

B. Performance

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

1) Secure Computation Circuits: We present secure computation building blocks in Table II. 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 $\mathcal{C}_{\text{XHTS}}$, $\mathcal{C}_{\text{k}^{m_1},\text{iv}}$ derive session secrets in milliseconds and compute CBs via the circuit $\mathcal{C}^{\text{X}}_{\text{ECB}_{2+}}$ 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 [3], [10] propose a scalable OT-based computation of

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

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

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

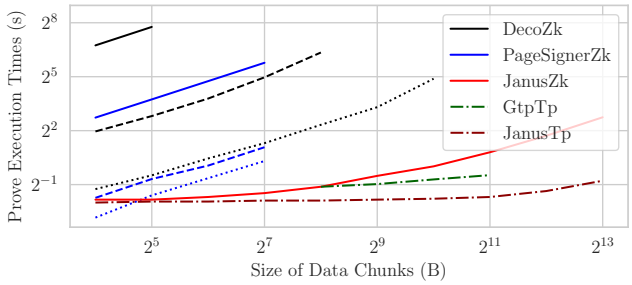


Fig. 8. Scalability analysis of transparent and privacy-preserving data openings, where solid, dashed, and dotted lines indicate zero-knowledge proof computations based on the *groth16*, *plonk*, and *plonkFRI* backends respectively. Dash-dotted lines indicate transparent data openings which are independent of zero-knowledge proof systems. Lines closer to the lower right corner are better and prove more data in less time.

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} verifications inside the circuit (cf. Figure 7). 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 II), 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.

2) Scalability Trade-off: Due to the fact that the initial works [2], [4] achieved practical computation times during the TLS handshake and record phase, our scalability analysis sets the focus on post-record computations. To gain a fair comparison between related works and the *Janus* protocol, we re-implemented privacy-preserving opening modes of related works based on the *Janus* software stack. For Deco [4], the zkSNARK circuit to open private data (DecoZk) depends on the AES128 computation in the GCM mode. Our results in Figure 8 show that DecoZk scales linearly with respect to the size of the opened plaintext data. Even though the scalability of our DecoZk re-implementation scales better with the *Groth16* zero-knowledge proof system, a comparison between the *Janus* protocol and DecoZk using the *plonkFRI* proof system is more suitable. The reason is that the *plonkFRI* proof system, in the same way as the *Janus* protocol, does not require a trusted setup protocol. We open source our AES128 GCM circuit implementation[4] as the first *gnark* circuit of AES. Our AES implementation performs worse compared to benchmarks found in the work [4]. This behavior is expected because our AES implementation uses a naive s-box lookup which related implementations optimize [28]. Thus, compared to DecoZk configured with the *plonkFRI* proof system, the *Janus* protocol opens private data 382x faster.

For the re-implementation of the zkSNARK circuit described by the work [3], we computed a zkSNARK friendly commitment on the 2PC output wire labels of the circuit $\mathcal{C}_{\text{CB}_{2+}}$. With the *mpc* framework [26], where the circuit encryption function encrypts 128-bit wire keys, committing to a 16 byte chunk of private TLS data requires a commitment on 2 kB of wire keys at the circuit output. With computational resources of the MacBook Pro, we were able to evaluate a MiMC commitment in the *gnark* circuit PageSignerZk on a maximum of 128 bytes (cf. blue lines in Figure 8). Thus, compared to a private data opening according to the Pagesigner protocol [3], the *Janus* protocol opens private data 3.4x faster. Last, we compare the transparent mode of the *Janus* protocol against transparent opening benchmarks presented in the work [12]. Compared to the Garble-Then-Prove approach (GtpTp) [12], we verify transparent data presentations 2.3x faster.

3) 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*. Notice that in the handshake phase, the circuit $\mathcal{C}_{\text{XATS}}$ must be called twice to compute SHTS and CHTS. The end-to-end benchmarks build upon the benchmarks distribution of our measured secure computation building blocks of Table II and the TLS 1.3 baseline reference. Additionally, we measure 321 milliseconds for the integrated 3PHS and a proceeding ECTF conversion. The local SHTS and certificate verifications in the handshake phase take 9.26

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

milliseconds and the witness pre-processing of the privacy-preserving *Janus* mode takes 4.47 milliseconds.

The resulting end-to-end benchmarks show that the computation overhead of the *Janus* protocol mainly affects the record phase which can be linked to the overhead introduced by the maliciously secure 2PC computation of requests. Regarding private data openings, 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 related works.

VII. Discussion

The discussion presents related works, positions the *Janus* protocol in the context of TLS oracles according to its benchmarks and security properties, and summarizes remaining limitations as well as future work directions.

A. Related Works

The garble-then-prove paradigm of the work [12] introduces semi-honest 2PC based on authenticated garbling to improve efficiency of TLS oracles. In contrast, TLS 1.3 allows the mutual authenticity verification of the SHTS secret if the *proxy* honestly garbles the circuit $\mathcal{C}_{\text{SHTS}}$. The *proxy* has the incentive to correctly garble $\mathcal{C}_{\text{SHTS}}$, because, with access to an authentic SHTS secret, the *proxy* can verify 2PC input correctness by matching 2PC inputs against SHTS. Thus, our work is able to deploy a semi-honest proof system based on 2PC without authenticated garbling, which increases the efficiency to verify private data. To prevent any information leakage, our work introduces a new commit-disclose-open paradigm. Related works, without access to a 2PC proof system leverage zkSNARK proof systems to verify private data as follows.

The work [11] 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 [11] 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 cipher verification in a dedicated 2PC proof system which can efficiently verify non-algebraic structures.

The works [2], [3] decouple the maliciously secure 2PC evaluation of response CBs through $\mathcal{C}_{\text{CB}}$, where the *client* obtains a 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 $\mathcal{C}_{\text{CB}}$, 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 [3]. 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 access plaintext data before a selection of data chunks is proven.

B. Positioning of the Janus Protocol.

The most recent work on efficient TLS oracles [12] achieves TLS 1.2 and TLS 1.3 compatibility and converts authenticated garbling parameters into a zkSNARK efficient Pedersen commitment. However, the private opening of TLS data against Pedersen commitments has not been evaluated. Thus, concerning TLS 1.3, we can only assume that the *Janus* protocol is ahead of the competition because our work does not depend on an extra conversion algorithm and a subsequent zkSNARK-based opening.

Due to the fact that our work achieves fast execution times with low communication cost in the handshake and record phase (cf. Table III), the *Janus* protocol is an interesting candidate to prove data of longer TLS 1.3 sessions. With the *Janus* protocol, users can efficiently collect TLS 1.3 data in long TLS sessions, and, in the end, selectively and efficiently decide which data to open. Further, due to the efficient privacy-preserving opening benchmarks, our work appears as a good candidate to attest to guaranteed TLS 1.3 delivery of larger data objects such as confidential documents. With the *Janus* protocol design, where the *verifier* acts as a *proxy*, our protocol integrates seamlessly into the web ecosystem as standard browsers allow the configuration of proxies. And, since the proxy mode of the work [4] does not provide *MITM-resistance*, the *Janus* protocol is the first proxy-mode TLS oracle with *MITM-resistance*.

C. Limitations & Future Work

The current version of the *Janus* protocol is tailored to the conditions found in TLS 1.3 and, as a consequence, is compatible with TLS 1.3 only. The second limitation is that in the current state of our implementation, the unoptimized computation of the request authentication tag includes algebraic structures which negatively impact the performance. Last, due to the fact that the related works [3], [10], [12] withhold benchmarks of privacy-preserving TLS data openings, our performance results can only be compared against our re-implementations.

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 refer to developing an efficient 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 the TLS 1.2 handshake secret can be leveraged to employ a HVZK proof system after introducing the asymmetric privacy setting. 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 1.3 employs encryption protocols which do not count as *key-binding* [11]. As a result, maliciously secure computation is used to compile requests and responses by computing the ciphertext and authentication tag [11]. In TLS 1.2, encryption algorithms are *key-binding* such that approaches only compute authentication tags using maliciously secure computation [2], [4]. Thus, the stronger cryptographic guarantees of TLS 1.2 encryption algorithms must be investigates with regard to introducing the asymmetric privacy setting with the secret disclosure of the *proxy*. We consider the investigation of the *Janus* paradigm in the context of TLS 1.2 as future work.

VIII. Conclusion

In this work, we reconsider the selection of secure computation techniques in TLS oracles by putting an emphasis on the conditions found in TLS 1.3. We find that the adversarial behavior of the TLS client-side can be partly reduced depending of the protocol phase of TLS. In our setting, where the verifier acts as a *proxy*, we show that the SHTS secret can be mutually verified if the *proxy* honestly garbles the semi-honest 2PC evaluation of SHTS. Even if the *proxy* acts maliciously during the computation of SHTS, the TLS security guarantees between the *server* and the *client* hold against the *proxy*. Further, for the case where the *proxy* honestly garbles the 2PC circuit of SHTS, the *proxy* can use the *server*-side authenticity of the SHTS parameter to verify the correctness of *client* inputs during 2PC protocols. This observation allows the deployment of a semi-honest proof system to privately or transparently convince the *proxy* of data provenance. The employed proof system achieves new standards for privacy-preserving and transparent data proofs.

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] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Y. Lindell, "Building mpc wallets – challenges and solutions," 2022.

[10] 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.

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

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

[13] 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.

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

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

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] M. Rossi, "Secure Multi-Party Computation (MPC) with Go," https://github.com/markkurossi/mpc, 2023.
[27] 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
[28] 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.
[29] 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.
[30] C. Reitwiessner, "zksnarks in a nutshell," *Ethereum blog*, vol. 6, pp. 1–15, 2016.
[31] 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.

APPENDIX

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.

A. Three-party Handshake

In the 3PHS (cf. Figure 9), 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 9, we show a TLS 1.3-specific configuration based on the ECDHE, where the parameters (e.g. Z_p) are EC points structured as $P = (x, y)$.

B. Digital Signatures

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

- **ds.Setup**(1^λ) $\rightarrow$ (sk, pk) takes in a security parameter λ and outputs a public key cryptography key pair (sk, pk).
- **ds.Sign**(sk, m) $\rightarrow$ (σ) takes in a secret key sk and message m and outputs a signature σ.
- **ds.Verify**(pk, m, σ) $\rightarrow \{0, 1\}$ 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.

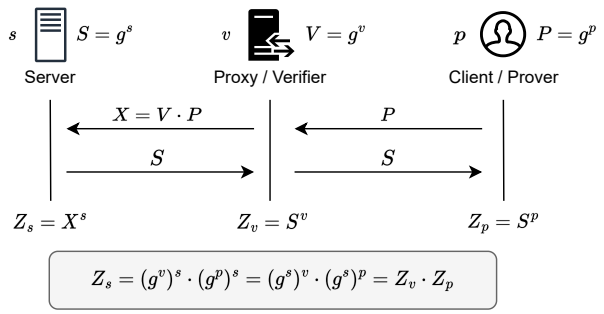


Fig. 9. Illustration of the 3PHS and exchanged cryptographic parameters between the server, the proxy, and the client. The gray box at the bottom indicates the relationship between shared client-side secrets Z_v and Z_p, which corresponds to the session secret Z_s of the server.

C. 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_{salt}, k_{ikm}) $\rightarrow$ (k_{pr}) takes in a string s_{salt}, input key material k_{ikm}, and returns a pseudorandom key k_{pr}.
- **hkdf.exp**(k_{pr}, s_{info}, l) $\rightarrow$ (k_{okm}) takes in a pseudorandom key k_{pr}, a string s_{info} and a length parameter l and returns output key material k_{okm} of length l.

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

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

D. 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 $\varnothing$.

E. 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 [15]. 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.

1) MtA Conversion based on Homomorphic Encryption: The secure 2PC MtA protocol converts multiplicative shares x, y into additive shares α, β such that $\alpha+\beta = x \cdot y = r$ yield the same result r. The MtA protocol exists in a vector form, which maps two vectors $\mathbf{x}, \mathbf{y}$, with a product $r = \mathbf{x} \cdot \mathbf{y}$, to two scalar values α, β, where the sum $r = \alpha + \beta$ is equal to the product r. The functionality of the vector MtA scheme can be instantiated based on Paillier additive Homomorphic Encryption (HE) [29]. 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 [22], 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) $\rightarrow$ (**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 **c1**=$[E_{sk_P}(x_1), \ldots, E_{sk_P}(x_l)]$.
- **mta.Eval**(**c1**,$\mathbf{y}$,pk_P) $\rightarrow$ (c_2,β) takes in the vector of ciphertexts **c1**=$[\mathrm{c1}_1, \ldots, \mathrm{c1}_l]$, 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 $\mathrm{c2} = \mathrm{c1}_1^{y_1} \cdot \ldots \cdot \mathrm{c1}_l^{y_l} \cdot E_{pk_P}(\beta')$ 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 **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.

2) ECTF Conversion: The ECTF algorithm is a secure 2PC protocol and converts multiplicative shares of two EC x-coordinates into additive shares [2], [4]. Figure 10 shows the computation sequence of the ECTF protocol which makes use the vector MtA algorithm defined in Section E1. 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 [2], [4]. 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.

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^λ); 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: α=**mta.Dec**(c2,sk);δ_1=$-x_1 \cdot \rho_1$+α;δ=δ_1+δ_2;η_1=$\rho_1 \cdot \delta^{-1}$; **c1**=**mta.Enc**($[-y_1$,$\eta_1]$,sk); send (**c1**,δ_1) to p_2
p_2: δ=δ_1+δ_2; η_2=$\rho_2 \cdot \delta^{-1}$; (c2, β)=**mta.Eval**(**c1**,$[\eta_2$,$y_2]$,pk); λ_2=$y_2 \cdot \eta_2$+β; send c2 to p_1
p_1: α=**mta.Dec**(c2,sk); λ_1=$-y_1 \cdot \eta_1 + \alpha$; **c1**=**mta.Enc**($[\lambda_1]$,$sk$); send **c1** to p_2
p_2: (c2,β)=**mta.Eval**(**c1**, $[\lambda_2]$, pk); $s_2 = 2 \cdot \beta + \lambda_2^2 - x_2$; send c2 to p_1
p_1: α=**mta.Dec**(c2,sk); $s_1 = 2 \cdot \alpha + \lambda_1^2 - x_1$

Fig. 10. 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 $P_1 + P_2$.

3) Oblivious Transfer: Secure 2PC based on boolean GCs depends on the 1-out-of-2 OT_2^1 sub protocol to secretly exchange input parameters of the circuit [17]. 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) $\rightarrow$ ($\mathrm{pp_{OT}}$) takes as input a security parameter λ and outputs public parameters $\mathrm{pp_{OT}}$ 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**($\mathrm{pp_{OT}}$) $\rightarrow$ (X) takes in $\mathrm{pp_{OT}}$, samples $x \xleftarrow{\$} \mathbb{Z}_p$, and outputs an encrypted secret $X = E_1(x)$.
- **ot.TransferY**($\mathrm{pp_{OT}}$, X, b) $\rightarrow$ (Y, k_D) takes in $\mathrm{pp_{OT}}$, 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 $b \stackrel{?}{=} 0$, or as $Y = X \cdot E_1(y)$ if $b \stackrel{?}{=} 1$.
- **ot.Encrypt**($\mathrm{pp_{OT}}$, X, Y, m_1, m_2, x) $\rightarrow$ (**Z**) takes in $\mathrm{pp_{OT}}$,

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}, $\mathbf{Z}$, 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**.

4) Semi-honest 2PC *with Garbled Circuits:* We define secure 2PC based on boolean garbled circuits by extending our OT definition of Section E3 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}) $\rightarrow$ (l, $T_{\text{l-w}}, T^G_{\text{l-w}}$, $T_{d\text{-}l}$) takes as input pp_{GC}, a boolean circuit $\mathcal{C}_G$, the input bit string d_{in}, and randomly samples $r_G, l \xleftarrow{\$} \mathbb{Z}_n$. The output consists of labels l, the undistorted and garbled label-to-wire tables $T_{\text{l-w}}, T^G_{\text{l-w}}$, and the bit-to-label table $T_{d\text{-}l}$.
- **gc.Evaluate**(pp_{GC}, $l_{\text{in}}^{d_{\text{in}}^{p_1}}$, $l_{\text{in}}^{d_{\text{in}}^{p_2}}$, $T^G_{\text{l-w}}$, $T^{\text{dec}}_{l_{\text{out}}\text{-}d_{\text{out}}}$) $\rightarrow$ (d_{out}) takes in pp_{GC}, input labels $l_{\text{in}}^{d_{\text{in}}^{p_1}}$, $l_{\text{in}}^{d_{\text{in}}^{p_2}}$, the garbled label-to-wire table $T^G_{l\text{-}w}$, a label-to-bit decoding $T^{\text{dec}}_{l_{\text{out}}\text{-}d_{\text{out}}}$, and outputs the bit string d_{out}.

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**, where p_1 loops over every wire of the circuit $\mathcal{C}_G$ and assigns two randomly generated labels to every wire. Each label pair at a wire correspond to a (0,1) bit pair such that in the end, the tables $T_{d\text{-}l}$ and $T_{l\text{-}w}$ provide a bit-to-wire mapping of all possible bit combinations. Further, the wires w and with that table $T_{l\text{-}w}$ encode the computation logic of $\mathcal{C}_G$. Next, p_1 uses the randomness r_G to turn $T_{l\text{-}w}$ into a garbled representation $T^G_{l\text{-}w}$, and determines labels $l_{\text{in}}^{d_{\text{in}}^{p_1}}$ at input wires according to input bits $d_{\text{in}}^{p_1}$. Subsequently, p_1 sends $l_{\text{in}}^{d_{\text{in}}^{p_1}}$,$T^{\text{dec}}_{l_{\text{out}}\text{-}d_{\text{out}}}$,$T^G_{l\text{-}w}$ to p_1.

Next, to obtain the remaining input labels $l_{\text{in}}^{d_{\text{in}}^{p_2}}$, p_1 and p_2 interact with the OT_2^1 scheme defined in Section E3. Because, p_1 is not allowed to learn the private input $d_{\text{in}}^{p_2}$ of p_2. Hence, per input bit of the string $d_{\text{in}}^{p_2}$, p_1 sends labels encrypted by **ot.Encrypt** as messages (m_1=$\hat{l}_{\text{in}}^{d_{\text{in}}^{p_2}}$, m_2=$\hat{l}_{\text{in}}^{\neg d_{\text{in}}^{p_2}}$) to p_2. Party p_2 obtains labels $l_{\text{in}}^{d_{\text{in}}^{p_2}}$ =$m_{d_{\text{in}}^{p_2}}$ by calling **ot.Decrypt**, and, with that, p_1 does not learn any bits $d_{\text{in}}^{p_2}$. With access to the labels $l_{\text{in}}^{d_{\text{in}}^{p_1}}$, $l_{\text{in}}^{d_{\text{in}}^{p_2}}$, p_2 is able to evaluate $T^G_{l\text{-}w}$ inside **gc.Evaluate** and, with the label-to-bit decoding table $T^{dec}_{l_{d_{\text{out}}}\text{-}d_{\text{out}}}$, p_2 translates computed output labels back to the bit string d_{out}.

5) Cryptographic Commitments: Cryptographic commitments are used to construct commitment schemes, which we define by the following tuple of algorithms. The function

- **c.Commit**(x) $\rightarrow$ (c, w) takes as input the data x and outputs a commitment string c and a witness w.
- **c.Open**(w, x, c) $\rightarrow$ $\{0, 1\}$ takes as input the witness w, the committed data x, and the commitment c and outputs a one if the only valid pair (w, x) is provided as input.

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.

F. 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.

1) Zero-Knowledge Succinct Non-Interactive Argument of Knowledge: A zkSNARK proof system is a zero-knowledge proof system, where the four properties of *succinctness*, *non-interactivity*, computational sound *arguments*, and witness *knowledge* hold [30]. 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 [31]), 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.

The security analysis audits the properties of *session-integrity*, *session-confidentiality*, and *session-authenticity* under the assumption that MITM attacks are feasible. Other system goals result from how the *Janus* protocol is designed. For our theorems, we assume that the security guarantees provided by TLS 1.3 [13] hold.

G. Handshake Phase

The *Janus* protocol ensures that in the TLS handshake phase the properties of *session-confidentiality*, *session-integrity*, and SHTS and *session-authenticity* hold.

1) Session Confidentiality: In the handshake phase, semi-honest 2PC circuits are executed as follows. Before evaluating the circuit $\mathcal{C}_{\text{XHTS}}$ with labels that output the secret SHTS, the *client* expects the authenticated and encrypted server-side messages SF, SC, and SCV and discloses the 2PC

output SHTS if the verification of SHTS against the server-side messages succeeds. The evaluation of $\mathcal{C}_{\text{XHTS}}$ for CHTS happens after the exchange of SHTS. Notice that CHTS is never shared with the *proxy*. Further, if the *proxy* honestly garbles $\mathcal{C}_{\text{SHTS}}$ and passes server-side messages back to the *client*, then the SHTS verification at the *client* succeeds and the correct SHTS value does not reveal anything about the HS secret due to the key independence property of TLS 1.3 [24]

Theorem 1. *Under a secure OT, garbling, and hash scheme, the semi-honest* 2PC *evaluation of $\mathcal{C}_{XHTS}$ with subsequent TLS 1.3-specific computations achieves session-confidentiality against a probabilistic polynomial time adversary $\mathcal{A}$ which maliciously garbles $\mathcal{C}_{XHTS}$, replays previously established handshake transcripts, and deviates during the* 2PC *evaluation.*

Proof 1. SHTS: If the adversary as a *proxy* performs a replay attack and shares previously obtained and authenticated handshake messages in a new *Janus* session, then the adversary can maliciously garble $\mathcal{C}_{\text{SHTS}}$ to output a SHTS' which matches the replayed server-side parameters. However, even if the *client* shares the 2PC output of SHTS' back to the adversary, SHTS' does not leak information of the *client*'s secret share. Since the *client* expects server-side messages before evaluating SHTS, SHTS' is already determined when the *client* obtains server-side messages. Hence, the adversary can only set SHTS' with a prediction on the *client* secret share. And, predicting the correct *client* secret share has negligible probability for the adversary. Thus, any malicious garbling of $\mathcal{C}_{\text{SHTS}}$ which leaks a secret bit of the *client*'s secret share yields an unpredictable outcome that is caught by the *client*'s SHTS verification against server-side messages[5]. If the *client* continues the TLS session after verifying SHTS' against a replayed transcript, then the *client* continues to use an uncompromised secret share. In that case, the *Janus* protocol ensures that the *client* eventually aborts the protocol before any secrets or confidential TLS data can be extracted by the adversary. Our protocol protects *client* secrets in the record phase through the deployment of maliciously secure computations techniques (cf. Appendix H1), or lets *clients* detect malicious *proxies* after the disclose of TLS session secrets (cf. Appendix H2).

CHTS: The semi-honest 2PC evaluation of the circuit $\mathcal{C}_{\text{CHTS}}$ is secure under the same cryptographic schemes as the evaluation of the circuit $\mathcal{C}_{\text{SHTS}}$. The secure OT and garbling scheme protect the *client* input secret share and CHTS is not disclosed to the adversary. Instead, the *client* follows the TLS specifications and derives hash-based keying material from CHTS to eventually disclose public AEAD parameters (ciphertext and authentication tag) to the adversary in form of AEAD protected handshake messages. A malicious garbling of $\mathcal{C}_{\text{CHTS}}$ leads from a protected propagation of CHTS=s_2 or CHTS=L_{s_2} to an indistinguishable set of AEAD parameters at the adversary. Thus, a semi-honest deployment of the circuit $\mathcal{C}_{\text{CHTS}}$ preserves *session-confidentiality*, because the adversary remains with a negligible probability of guessing s_2.

2) Session Integrity: Session-integrity in the handshake phase of the *Janus* protocol is only partly supported. Generally, *session-integrity* prevents an adversary from deviating the TLS 1.3 handshake specification and provides honest parties with an identifiable abort option when detecting misbehavior of the adversary. However, due to the ability of the adversary to create and present authentic traffic transcripts through replay and MITM attacks, *session-integrity* in the handshake phase only holds for the case where a single attack is false input provision.

Theorem 2. *Under an honest server, and a secret shared client session, the Janus protocol preserves session-integrity against a probabilistic polynomial time adversary $\mathcal{A}$ which deviates from the TLS 1.3 specification by providing false inputs into semi-honest* 2PC *computations.*

Proof 2. Injecting false inputs into the semi-honest 2PC circuit $\mathcal{C}_{\text{XHTS}}$ leads to a computation of SHTS' and CHTS', which deviate from server-side secrets SHTS and CHTS. As a consequence, the *server* is incapable of processing non-compliant AEAD parameters which have been derived under false session secrets. Further, the SHTS verification fails against authenticated and protected handshake messages of the *server*.

Notice. The adversary is able to modify traffic transcripts through replay attacks which cannot be detected by an honest party in the TLS handshake phase but will be eventually detected by the honest party during the post-record phase. Until the detection of the malicious adversary, the *Janus* protocol ensures *session-confidentiality*.

3) Session Authenticity: In the handshake phase, we investigate *session-authenticity* with regard to unforgeability of server-side PKI certificates and transcripts.

Theorem 3. *Under the honest server assumption and a secure digital signature scheme, the Janus protocol achieves session-authenticity against a probabilistic polynomial time adversary $\mathcal{A}$ which tries to forge a message transcript for a valid certificate.*

Proof 3. The adversary can only forge a TLS handshake transcript, if the adversary is able to compute the messages SF, SCV, and SC. Computing the messages SF, SCV, and SC requires the computation of a signature on the handshake message transcript, where the signature corresponds to the information of the certificate in the SC message. Since the adversary cannot obtain the private key which corresponds to the public key of the certificate, the adversary cannot forge a signature. Thus, *session-authenticity* is preserved for a message transcript if any client party verifies the *server*'s PKI certificate, authenticates the *server*'s signature in the SCV message, and checks a correct session transcript by verifying the SF digest.

[5]This reasoning contradicts the incorrect arguing of the work [10] which declares the *Janus* protocol vulnerable to the described replay attack by a malicious *proxy*.

Notice. Our system model allows replay attacks where adversaries can obtain *session-authenticity* on multiple transcripts. Agreeing on a cipher suite before instantiating a TLS 1.3 session, and setting a key share in the CH message allows the *server* compute all TLS 1.3 session secrets. Subsequently, the *server* derives session secrets and authenticates server-side handshake messages using the signature scheme. Since the *server* signs the message transcript of the obtained CH message and derives all other message transcripts locally, the adversary can only determine the structure of the CH message and obtain authenticated message transcripts which depend on a secret server randomness.

4) SHTS Authenticity: In the handshake phase, we investigate the authenticity of SHTS in the case of an honestly acting *proxy*, and consider the *client* as the adversary.

Theorem 4. *Under the honest server and proxy assumption, a secure digital signature, a secure OT and semi-honest garbling scheme scheme, the semi-honest* 2PC *evaluation and disclosure of SHTS achieves authenticity for the proxy against a probabilistic polynomial time adversary $\mathcal{A}$ which tries to forge the authenticity of SHTS for a given CH message.*

Proof 4. The *client* shares a CH message with the *proxy* where the *proxy* overwrites the key share value by adding another key share via the 3PHS. With the addition of the *proxy* secret, the CH message is fully determined and the *proxy* records the CH transcript first and before the *client* can predict the CH transcript hash. Further, the CH message is uniquely defined by the *client* randomness. To forge the authenticity of the SHTS value, the adversary must find another transcript of SF, and SCV according to SC, which is infeasible because the adversary cannot forge a signature according to the server certificate. If the adversary performs a replay attack, then the adversary can determine a SHTS' which matches authenticated messages. Instead of evaluating an honest 2PC of the SHTS circuit, the *client* simply shares SHTS' with the *proxy* such that the *proxy* verifies authenticity of SHTS'. However, the attack fails when the *proxy* decrypts the SF message and checks the transcript hash of the signature according to the CH message. The adversary is only able to request authenticated handshake messages for individual CH messages because the adversary cannot compute the signature otherwise. In addition, the *proxy* can verify if the right authenticated messages match the CH message because every CH message is individually defined by the *client* randomness which the *proxy* learns before the *client*. Thus, any detection of the CH transcript hash deviation leads to an abort by the *proxy*.

H. Record & Post-record Phase

The security analysis of the record phase investigates if the security properties of *session-confidentiality* and *session-integrity* hold throughout the TLS record and post-record phase. *Session-authenticity* holds in the *Janus* protocol if *session-authenticity* is preserved for parameters that achieve *session-integrity* in the post-record phase.

1) Session Confidentiality: In the record phase, the adversary tries to maliciously garble 2PC circuits such that shared outputs either reveal the secret input share of the *client* parties or yield information on AEAD parameters which leak private record data of the *client* (e.g. CBs to decrypt ciphertext chunks). Malicious *clients* try to learn secret shares of the *proxy* with the goal to forge compliant traffic transcripts via MITM attacks such that arbitrary data can be injected into the *Janus* protocol. For *clients*, *session-confidentiality* is supposed to hold throughout the *Janus* protocol whereas for the *proxy*, *session-confidentiality* applies until the disclosure phase where the *proxy*, according to the *Janus* protocol, shares all session secrets with the *client*.

Theorem 5. *Under a secure OT, commitment, and garbling scheme and a semi-honest proof system leaking a single bit, the Janus protocol maintains session-confidentiality during the record and post-record phases against a probabilistic polynomial time adversary $\mathcal{A}$, which maliciously garbles* 2PC *circuits, acts as a malicious verifier, or deviates as the evaluator in* 2PC *systems.*

Proof 5. For the adversary $\mathcal{A}_1$, acting as a *client*, the interaction to compute all record and post-record circuits $\mathcal{C}_{(\mathrm{k}^{m_1},\mathrm{iv})}$, $\mathcal{C}_{\mathrm{ECB}_{2+}}$, $\mathcal{C}_{tag}$, $\mathcal{C}_{\mathrm{tpOpen}}$, and $\mathcal{C}_{\mathrm{zkOpen}}$ never leaks secret information of the *proxy* due to the secure OT and garbling scheme. Thus, $\mathcal{A}_1$ never learns secret shares of the *proxy*. For an adversary $\mathcal{A}_2$, acting as the *proxy*, no 2PC interactions to compute the circuits $\mathcal{C}_{(\mathrm{k}^{m_1},\mathrm{iv})}$, $\mathcal{C}_{\mathrm{ECB}_{2+}}$, $\mathcal{C}_{tag}$ leaks a bit due to the deployment of the maliciously secure 2PC system in the record phase. In the post record phase, the adversary $\mathcal{A}_2$ does not learn any bit throughout the semi-honest interaction in the proof system to compute $\mathcal{C}_{\mathrm{zkOpen}}$, because the commit-open-disclose paradigm yields a commitment opening after the decoding table of the circuits has been shared with the *client*. The secure commitment does not leak any information on the output wire keys to the adversary $\mathcal{A}_2$. As a result, the *client* is able to detect any incorrect garbling that would leak private information and aborts the protocol upon detecting any leakage. During the evaluation of the circuit $\mathcal{C}_{\mathrm{tpOpen}}$, *session-confidentiality* is not preseved anymore because $\mathcal{A}_2$ learns the data which is transparently shared.

2) Session Integrity: To compromise *session-integrity* in the record or post-record phase, the adversary intends to inject arbitrary AEAD parameters into the interactive computation of record ciphertexts and authentication tags. Further, the adversary tries to convince the verifier from an incorrect mapping of plaintext chunks during the semi-honest interaction in the proof system. The adversary continues to leverage the sharing of false 2PC outputs or provides false inputs to 2PC circuits in order to deviate from the protocol.

Theorem 6. *Under a semi-honest proof system leaking a single bit, a secure OT, garbling, and commitment scheme, and a maliciously secure* 2PC *system, the Janus protocol preserves session-integrity against a probabilistic polynomial time adversary $\mathcal{A}$ which acts as a cheating prover, a malicious garbler,*

or injects arbitrary TLS parameters or data throughout the protocol.

Proof 6. The adversary $\mathcal{A}_1$, acting as a *client*, cannot inject arbitrary response records into the *Janus* protocol because the adversary $\mathcal{A}_1$ only obtains full session secrets to compute valid responses after the *proxy* stops the recording of record transcripts. Further, since the *proxy* verifies every selected transcript record pair ($\mathbf{x}$, t^x), which has been selected by the adversary, in the semi-honest proof system, the adversary $\mathcal{A}_1$ cannot inject arbitrary response data into the *Janus* protocol. The adversary $\mathcal{A}_1$ can neither inject request transcripts that pass the verification at the *server* and that are not recorded at the *proxy*, because the *proxy* only participates in the 2PC computation of request authentication tags if the corresponding ciphertext has been shared. In the post-record phase the integrity of correct data provenance is verified by the *proxy* by requiring the adversary to present a valid proof in the semi-honest proof system. In the proof system, any false input provision of the adversary $\mathcal{A}_1$ for the circuits $\mathcal{C}_{\text{tpOpen}}$ and $\mathcal{C}_{\text{zkOpen}}$ is caught, because the honest *proxy* verifies input against an already authenticated SHTS parameter (cf. Section G4). As such the computation to be proven verifies intercepted and recorded traffic transcripts of the *proxy* against the correct input of the adversary.

If the adversary $\mathcal{A}_2$, taking the role of a malicious *proxy*, maliciously garbles the 2PC circuits $\mathcal{C}_{(\text{k}^{m_1},\text{iv})}$, $\mathcal{C}_{\text{ECB}_{2+}}$, and $\mathcal{C}_{tag}$, then the *client* detects the adversary through the maliciously secure 2PC system which is used to compute the circuits. The malicious garbling of the circuits and $\mathcal{C}_{\text{tpOpen}}$ and $\mathcal{C}_{\text{zkOpen}}$ is detected by the *client* after the *proxy* discloses the garbled truth tables in the commit-disclose-open paradigm. Further, we consider the adversary $\mathcal{A}_2$ to successfully pass the SHTS verification of the *client* in the handshake phase based on a replay attack. In that case, the *client* has verified a malicious SHTS'. The adversary is left with the options to (i) input arbitrary data into honestly garbled 2PC circuits, and (ii) disclose arbitrary data when disclosing all session secrets. With (i), the adversary can compute and accept arbitrary requests from 2PC request computations and, according to the forged TLS session, compute and share valid responses with the *client*. The arbitrary encryption of records prevent the adversary from accessing any secrets. However, with (ii), the adversary is forced to share a session secret which, combined with the secret share of the *client*, must yield a SHTS'$\stackrel{?}{=}$SHTS in a honest and local computation of SHTS. The adversary only knows the underlying secret that leads to the computation of SHTS' and cannot predict the information preserving addition of secret shares without knowledge of the *client* secret share. Thus, the disclosure requirement of all session secrets leads to the detection of a cheating $\mathcal{A}_2$ because the comparison of the local computation of SHTS against the previously obtained SHTS' cannot be predicated by the adversary and leads to a mismatch.

During the evaluation of $\mathcal{C}_{\text{tpOpen}}$ and $\mathcal{C}_{\text{zkOpen}}$, only the *client* inputs private data such that the adversary $\mathcal{A}_2$ can only deviate form the *Janus* protocol if $\mathcal{A}_2$ maliciously garbles the circuits. As already stated in the previous sentences, a malicious garbling of $\mathcal{A}_2$ in the record and post-record phase is infeasible.