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 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 commitment-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.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_2$</td>
<td>$H(\text{ClientHello}</td>
</tr>
<tr>
<td>$H_3$</td>
<td>$H(\text{ClientHello}</td>
</tr>
<tr>
<td>$H_6$</td>
<td>$H(\text{ClientHello}</td>
</tr>
<tr>
<td>$H_7$</td>
<td>$H(\text{ClientHello}</td>
</tr>
<tr>
<td>$H_9$</td>
<td>$H(\text{ClientHello}</td>
</tr>
<tr>
<td>label$_{11}$</td>
<td>“TLS 1.3, server CertificateVerify”</td>
</tr>
</tbody>
</table>

$(k_{\text{CATS}}, iv_{\text{CATS}}) \mid (k_{\text{CATS}}, iv_{\text{CATS}})$

DeriveTK($s=SATS \mid CHTS$) =

$= (\text{hkdf}.exp(s, “key”, H(“”), len(k)), \text{hkdf}.exp(s, “iv”, H(“”), len(iv)))$

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 $\text{len}(\mathbf{x}) = n$ returns the length of the vector. Base points of elliptic curves are represented by $G \in \text{EC}(\mathbb{F}_p)$, where the finite field $\mathbb{F}_p$ is of a prime size $p$. For elliptic curve elements, the operators $\cdot, +$ refer to the scalar multiplication and addition of elliptic curve points $P \in \text{EC}(\mathbb{F}_p)$. The symbol $\lambda$ 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 $\overset{\$}{\leftarrow}$, the concatenation of strings with $||$, and the comparison of variables with $\approx$.

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. $\text{ds.Sign}$, $\text{ds.Verify}$, $\text{hkdf.ext}$, $\text{hkdf.exp}$, $\text{hmac}$ 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) 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 $m_{\text{CH}}, m_{\text{SH}}$. 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

- 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 B), provide end-to-end evaluation benchmarks (cf. Section VI), and open-source the implementation of our secure computation building blocks.
next, continues to use the derived record parameters.

The TLS record phase, which we side and client-side application traffic keys and initialization traffic secret (SATS), which are used to compute server-side application traffic keys. 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 into a confidential and authenticated representation (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 A4. 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 p1 with input x as the garbler and party p2 with input y as the evaluator. Party p1 is supposed to generate the garbled circuit G(C), where the boolean circuit C implements the logic of the public function f (cf. Figure 4). To generate the garbled circuit, p1 randomly samples wire keys k^x_i, k^y_i and a signal bit σ_L at every wire w_L. For the purpose of evaluating the function f, wire keys k^y_i encode binary data representations of f using internal labels i. The purpose of signal bits is twofold. Signal bits encrypt internal bits to external bits e_L = σ_L ⊕ i which can be shared with p2. With that, signal bits enable the evaluator to discover valid entries of garbled tables G(C) through external bits e [16]. Further, signal bits randomize garbled truth tables G(C) to obfuscate truth table bit mappings.

Once wire keys, signal bits, and external labels exist, p1 computes the garbled table entries as follows. Per row of table G(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 \( G(C), T_{1\rightarrow d} \), and, if \( x=[1,0], (k_{0^L}, e=0) \) and \( (k_{0^R}, e=1) \) 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 \( k_{0^L}, k_{0^R} \) with corresponding external values with \( p_2 \). Further, the OT scheme gives \( p_2 \) access to the keys \( (k_{0^L}, e=0) \) and \( (k_{0^R}, e=0) \) if \( y=[0,1], \) and prevents \( p_1 \) from learning \( p_2 \)’s selection of wire keys. With access to \( G(C) \), input wire keys and corresponding external labels, \( p_2 \) is able to evaluate the garbled circuit. To evaluate the first output bit, \( p_2 \) decrypts the third entry of table \( G(C_{AND}) \) and obtains \( (k_{0^L}, e=0) \). With that, \( p_2 \) continues to decrypt the first entry of table \( G(C_{AND}) \) to obtain \( k_{0^L} \) (cf. Figure 4). Last, \( p_2 \) decodes \( k_{0^L} \) using the decoding table \( T_{1\rightarrow d} \) 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\(_2\) sub protocol to secretly exchange input parameters of the circuit [17]. The OT\(_2\) 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 \( m_0 \) is revealed to party \( p_2 \). Party \( p_2 \) inputs a secret bit \( b \) which decides the selection of the message \( m_0 \). In this work, we make use of the OT\(_2\) 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 \( k_x \) and output bits \( b \) from a correctly garbled circuit of \( p_2 \), then \( p_1 \) knows which output labels \( k_y \) 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 \( H(k_y) \) with \( p_2 \) after the first circuit evaluation, and \( p_2 \) returns the same hash \( H(k_y) \) 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 \( k'_x \) and \( k'_y \) and the correct bits \( b \) lead \( p_2 \) to a correct selection of \( k_y \neq k'_y \). Since \( p_2 \) does not know which output keys \( p_1 \) evaluates, \( p_2 \) cannot predict any keys \( k'_x, k'_y \) 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 \( L \) and the existence of an algorithm \( R_L \), which decides in polynomial time if \( w \) is a valid proof for the statement \( x \in L \) by evaluating \( R_L(x, w) = 1 \). The assumption is that for any statement \( x \in L \), there exist a valid witness \( w \) and no witness exists for statements \( x \notin L \). 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 A1 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 present further details of the ECTF algorithm in the Appendix A5b. 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 kdf.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 $C_{SHTS}$ 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 $C_{SHTS}$ against authenticated handshake messages, then no secret of the client can be extracted by the proxy (cf. Appendix B1a).

If the proxy honestly garbles the 2PC circuit $C_{SHTS}$ and verifies the respective SHTS output against authenticated messages of from the server, then authenticity for SHTS is guaranteed (cf. Appendix B1d). 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, 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 $C_{CHTS}$ 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 B1a).

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 $C_{CB}$ 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 $C_{tag}$, because the authentication tag is made public as a part of the request.

---

**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.
record. If the proxy maliciously garbles $C_{tag}$ 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-confidence is compromised. Additionally, the maliciously secure 2PC evaluation of $C_{tag}$ 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 AAD verification. The maliciously secure computation of request CBs prevents the following case. If the proxy maliciously garbles $C_{CB}$ to output known CBs and honestly garbles $C_{tag}$, then the client cannot detect false CBs. With knowledge of CBs, the proxy is able to decrypt ciphertext chunks and break session-confidence.

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 $C_{zkOpen}$, 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 $C_{zkOpen}$ if the proxy has honestly garbled $C_{SHTS}$ and successfully verified SHTS. If the $C_{zkOpen}$ 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 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 $\phi$ 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 A1). 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 A5b). 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 $C_{SHTS}$ 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 $C_{SHTS}$ 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 \( C_{\text{tag}} \), 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 \( C_{\text{XHTS}} \) 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 \( C_{\{k=1, iv\}} \) 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 \( k_{\text{XATS}}^{m_1} = m_1 \otimes k_{\text{XATS}} \) and the request initialization vector \( iv_{\text{XATS}} \). 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 \( C_{\text{CB}^2+} \), which outputs counter blocks \( \text{CB}^2+ \) to the client. The notation \( \text{CB}^2+ \) expresses counter blocks \( \text{CB} \) with an index \( i > 1 \). With access to \( \text{CB}^2+ \), the client computes the ciphertext chunks \( q \) of the request record and shares \( q \) with the proxy. Once the proxy receives \( q \), both the proxy and client compute the 2PC circuit \( C_{\text{tag}} \) (cf. dark blue circuit in Figure 6) to complete the construction of the request. The circuit \( C_{\text{tag}} \) outputs the counter blocks \( \text{CB}_{\text{tag}} = [\text{CB}_0, \text{CB}_1] \), which, together with the ciphertext chunks \( q \), 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 \( C_{\text{tag}}, C_{\text{CB}^2+}, \) and \( C_{\{k=1, iv\}} \).

3) Record Response Circuits: After computing and sending requests to the server, the proxy records (i) request ciphertext and authentication tag pairs \( (q, t') \) and (ii) proceeding response ciphertext and authentication tag pairs \( (r, t') \) from the server. When processing server-side response data, the execution of the 2PC circuit \( C_{\text{ECB}^2+} \) 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 \( C_{\text{ECB}^2+} \) 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 \( r = [r_1, \ldots, r_l] \), where each response \( r_m \) has an index \( m < n \), then the client and proxy perform \( l \) executions of the circuit \( C_{\text{CB}^2+} \). The evaluation of \( l \) circuits \( C_{\text{CB}^2+} \) yields \( l \) vectors of encrypted counter blocks \( \text{CB}^2+ \) to the client. With \( l \) vectors of \( \text{CB}^2+ \), the client is capable of accessing the contents of the responses \( r = [r_1, \ldots, r_l] \) to construct the \( n \)-th request. Further, with access to the \( l \)-th \( \text{CB}^2+ \), the client is incapable of accessing any TLS session secret share of the proxy and, thus, cannot compute any \( y \)-th \( \text{CB}^2+ \), with \( y > l \). To preserve MITM-resistance, it must hold that the proxy intercepts the pair \( (r, t') \) before the circuit \( C_{\text{CB}^2+} \) outputs the corresponding \( \text{CB}^2+ \) of response \( r \). Notice that no \( \text{CB}^2+ \) vector includes any \( \text{CB}_{\text{tag}} \) parameters.

C. Post-record Verification

As soon as the client has exchanged enough record data with the server such that the statement \( \phi \) 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 $CB_{tag}$ to the proxy such that the proxy is able to verify the authenticity of recorded traffic. A traffic pair $(x, t^s)$ is authentic if the counter blocks $CB_{tag}$ and the ciphertext chunks $x$ compute the corresponding tags $t^s$. 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 $I^{\text{open}}$ and shares the list with the proxy. The list $I^{\text{open}}$ contains GCM indices of CB, ciphertext, and plaintext chunks. With $I^{\text{open}}$, the proxy can select required ciphertext chunks for the data provenance verification. Depending on the operation mode of the Janus protocol, the indices $I^{\text{open}}$ 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 $I^{\text{open}}$ 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 $I^{\text{open}}$ and proves knowledge of authentic plaintext data, and evaluates the plaintext against the public statement $\phi$ (cf. Section V-C3). Again, the list of indices $I^{\text{open}}$ determines which plaintext chunks are (i) verified concerning session-authenticity and session-integrity, and (ii) validated against the statement $\phi$.

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 verifies 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 $\phi$ is revealed to the proxy.

Transparent Data Opening In the transparent mode of Janus, the proxy garbles the semi-honest 2PC circuit $C_{tpOpen}$, 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 $T_{1-d}$ 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 $ct_{po}=\text{Com}(l_{\text{out}}^{\text{SHTS}}, k_{\text{out}}^{\text{CB}}, r)$ and returns the commitment $ct_{po}$ back to the proxy. After obtaining the commitment, the proxy shares the garbled truth tables $G(C_{tpOpen})$ including the output decoding tables $T_{1-d}$ with the client. The client verifies correct garbling of the $C_{tpOpen}$ circuit and shares the commitment randomness $r$ with the proxy if the $C_{tpOpen}$ circuit verification succeeds. The proxy uses the commitment randomness $r$ to open the commitment $ct_{po}$ 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 $CB_{tag}$ parameter must match the previously shared $CB_{tag}$ values (cf. Section V-C1).

Privacy-preserving Data Opening In the privacy-preserving mode, the proxy garbles the semi-honest 2PC circuit $C_{zkOpen}$ (cf. Figure 7) and withholds the decoding and truth tables $T_{1-d}, G(C_{zkOpen})$ from the client until the proxy receives the commitment $c_{zk}=\text{Com}(l_{\text{out}}^{C_{zk}} , r)$. 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 $c_{zk}$ maps to a 1. The difference between the privacy-preserving and the transparent data opening is that the circuit $C_{zkOpen}$ verifies $pt$ and the derived $CB$ against $ct$ as well as the parameters SHTS and $CB_{tag}$ 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 $\phi$. We express the statement compliance evaluation with the function call $f_{\phi}(pt)$ (cf. line 2 in Figure V-C2). To clarify

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 $f_{\phi}$ validates input data against a public statement $\phi$ and returns $a$ if the validation succeeds.
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 $\phi$ depends on the data inside the third ciphertext chunk of the first response ($r_1$, $r_2$), then the index=3 is included in the list of indices $I^{\text{open}}$. With $I^{\text{open}}$, the zkOpen circuit is able to compute the right $CB_{2+\text{index}}$, and consider $CB_3=\text{AES}(k, iv||\ldots|5)$ for the computation of $ct_5 = CB_5 \oplus pt_5$. If the assertions against public inputs succeed (e.g. $ct_5 \oplus = ct_5$), 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 $\phi$.

### D. Data Attestation

Once the client verification at the proxy succeeds, the proxy attests 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 $p_{\text{cert}}(t, \phi, pk, \sigma)$ of the transparent attestation includes a signature $\sigma = ds.Sign(sk, [\phi, t])$ computed at time $t$. The proxy overwrites the statement $\phi = H(pt)$ to the hash of verified data such that every third party can evaluate arbitrary statements against the attested and public data. With $p_{\text{cert}}$, the client gains public verifiability of TLS 1.3 data and can present $p_{\text{cert}}$ to any third party who trusts the proxy. In the privacy-preserving mode, the structure of $p_{\text{cert}}$ remains the same except that the statement $\phi$ 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 $\phi$ 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 (Ma) conversion algorithms in Golang 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 $C_{\text{XHTS}}$. $C_{\text{EBCB}_2+}$ derive session secrets in milliseconds and compute CBs via the circuit $C_{\text{EBCB}_2+}$ for a 2 kB record in 164.9 milliseconds. An interesting fact to notice is that the AED tag circuit $C_{\text{tag}}$ is efficient for small request sizes and scales sufficiently but not ideally for larger request sizes. The overhead in the circuit $C_{\text{tag}}$ 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.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Communication (kb)</th>
<th>Execution LAN (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Offline</td>
<td>Handshake</td>
</tr>
<tr>
<td>TLS 1.3</td>
<td>-</td>
<td>1.94</td>
</tr>
<tr>
<td>Janus\textsubscript{transparent}</td>
<td>305.17 (MB)</td>
<td>-</td>
</tr>
<tr>
<td>Janus\textsubscript{private}</td>
<td>406.29 (MB)</td>
<td>-</td>
</tr>
</tbody>
</table>

![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.](image)

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 $C_{\text{Open}}$ is more efficient compared to the privacy-preserving mode with the circuit $C_{\text{zkOpen}}$. This behavior is expected because, the 2PC circuit of the transparent mode does not include the ciphertext, SHTS, and CB\textsubscript{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. $f_\phi$ of the last row in Table II), we evaluate the verification of a confidential document hash $H(f)$ in the circuit $C_{\text{zkOpen}}$. To do so, we set the function $f_\phi = H(f)$ ? $H(\text{pt})$ 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\footnote{https://github.com/Consensys/gnark/pull/719} 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 $C_{\text{CB2+}}$. 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 $C_{\text{XATS}}$ 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
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 $C_{SHTS}$. The proxy has the incentive to correctly garble $C_{SHTS}$, 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 preprocessing 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 $C_{CB}$, 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 $C_{CB}$, 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.

The works [2], [3] decouple 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.
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 framework 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


A. Cryptographic Building Blocks

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

1) 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 · 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).

2) Digital Signatures: A digital signature scheme is defined by the following tuple of algorithms, where

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

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

3) Keyed-hashed or Hash-based Key Derivation Function: A HKDF function converts parameters with insufficient randomness into suitable keying material for encryption or authentications algorithms. The HKDF scheme is defined by a tuple of algorithms, where

- **hkdf.ext**(salt, k_km) → (kp) takes in a string salt, input key material k_km, and returns a pseudorandom key kp.
- **hkdf.exp**(kp, info, l) → (k_km) takes in a pseudorandom key kp, a string info and a length parameter l and returns output key material k_km 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).

\[
\text{hmac}(k, m) = H((k' \oplus \text{opad}) || H((k' \oplus \text{ipad}) || m))
\]

with \( k' = H(k) \), if len(k) > b

\[ k' = k, \text{ else} \]

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

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

5) Secure Two-party Computation: Secure 2PC allows two mutually distrusting parties with private inputs \( x_1, x_2 \) to jointly compute a public function \( f(x_1, x_2) \) without learning the private input of the counterparty. With that, secure 2PC counts as a special case of multi-party computation (MPC), with \( m = 2 \) parties and the adversary corrupting \( t = 1 \) parties [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.

a) MtA Conversion based on Homomorphic Encryption:
The secure 2PC MtA protocol converts multiplicative shares $x, y$ into additive shares $\alpha, \beta$ 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 $x, y$, with a product $r = x \cdot y$, to two scalar values $\alpha, \beta$, 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

- $\text{mta.Setup}(1^\lambda) \rightarrow (sk_p, pk_p)$ takes in the security parameter $\lambda$ and outputs a Paillier key pair $(sk_p, pk_p)$.
- $\text{mta.Enc}(x, sk_p) \rightarrow (c_1)$ takes in a vector of field elements $x = [x_1, \ldots, x_l]$ and a private key $sk_p$ and outputs a vector of ciphertexts $c_1 = [E_{sk_p}(x_1), \ldots, E_{sk_p}(x_l)]$.
- $\text{mta.Eval}(c_1, y, pk_p) \rightarrow (c_2, \beta)$ takes in the vector of ciphertexts $c_1 = [c_{11}, \ldots, c_{1l}]$, a vector of field elements $y = [y_1, \ldots, y_l]$, and a public key $pk_p$. The output is a tuple of a ciphertext $c_2 = c_1^{y_1} \cdot \cdots \cdot c_1^{y_l} \cdot E_{pk_p}(\beta')$ and the share $\beta = -\beta'$, where $\beta' \leftarrow Z_p$.
- $\text{mta.Dec}(c_2, sk_p) \rightarrow (\alpha)$ takes in a ciphertext $c_2$ and a private key $sk_p$ and outputs the share $\alpha = D_{sk_p}(c_2)$.

The tuple of algorithms is supposed to be executed in the order where party $p_1$ first calls $\text{mta.Setup}$ and $\text{mta.Enc}$. 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 $c_1$ with party $p_2$, then $p_2$ calls $\text{mta.Eval}$ and shares the ciphertext $c_2$ with $p_1$. Last, $p_1$ calls $\text{mta.Dec}$, where $D_k(z)$ is a Paillier decryption of message $z$ under key $k$. If the algorithms are executed in the described order, then party $p_1$ inputs private multiplicative shares in the vector $x$ and obtains the additive share $\alpha$. Party $p_2$ inputs the private vector of multiplicative shares $y$ and obtains the additive share $\beta$. In the end, the relation $x \cdot y = \alpha + \beta$ holds, and neither the party $p_1$ nor the party $p_2$ learn anything about the private inputs of the counterparty.

b) ECTF Conversion: The ECTF algorithm is a secure 2PC protocol and converts multiplicative shares of two EC coordinates into additive shares [2], [4]. Figure 10 shows the computation sequence of the ECTF protocol which makes use of the vector MtA algorithm defined in Section A5a. By running the ECTF protocol, two parties $p_1$ and $p_2$, with EC points $P_1$, $P_2$ as respective private inputs, mutually obtain additive shares $s_1$ and $s_2$, which sum to the $x$-coordinate of the EC points sum $P_1 + P_2$. TLS oracles use the ECTF protocol to transform the client-side EC secret shares $Z_1$ and $Z_2$ 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.

c) Oblivious Transfer: Secure 2PC based on boolean GCs depends on the 1-out-of-2 OT$_2^b$ sub protocol to secretly exchange input parameters of the circuit [17]. The OT$_2^b$ 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

- $\text{ot.Setup}(1^\lambda) \rightarrow (pp_{\text{OT}})$ takes in input a security parameter $\lambda$ and outputs public parameters $pp_{\text{OT}}$ 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.
- $\text{ot.TransferX}(pp_{\text{OT}}) \rightarrow (X)$ takes in $pp_{\text{OT}}$, samples $x \leftarrow Z_p$, and outputs an encrypted secret $X = E_1(x)$.
- $\text{ot.TransferY}(pp_{\text{OT}}, X, b) \rightarrow (Y, k_D)$ takes in $pp_{\text{OT}}$, a cipher $X$, a bit $b$, and samples $y \leftarrow Z_p$. The output is a decryption key $k_D = X^y$ and a cipher $Y$ encrypting as $Y = E_1(y)$ if $b = 0$, or as $Y = X \cdot E_1(y)$ if $b = 1$.
- $\text{ot.Encrypt}(pp_{\text{OT}}, X, m_1, m_2, x) \rightarrow (Z)$ takes in $pp_{\text{OT}}$, $Y$, and derives $k_1 = H(Y^x)$, $k_2 = H((Y^x)^y)$. The output is a vector of ciphers $Z = [E_2(m_1, k_1), E_2(m_2, k_2)]$.
- $\text{ot.Decrypt}(pp_{\text{OT}}, Z, k_D, b) \rightarrow (m_b)$ takes in $pp_{\text{OT}}$, key $k_D$, the bit $b$, and a vector of ciphers $Z = [Z_1, Z_2]$. The output is the message $m_b = D_2(Z_b, k_D)$.

In the OT$_2^b$ protocol, party $p_1$ calls $\text{ot.Setup}$ and $\text{ot.TransferX}$, and sends the public parameters and cipher $X$ to $p_2$. Party $p_2$ calls $\text{ot.TransferY}$, locally keeps the decryption key and shares the cipher $Y$ with $p_1$. Now, $p_1$ shares the output of
ot.Encrypt with $p_2$, who obtains $m_b$ by calling ot.Decrypt.

d) Semi-honest 2PC with Garbled Circuits: We define secure 2PC based on boolean garbled circuits by extending our OT definition of Section A5c with the tuple of algorithms, where

- \textbf{gc.Setup}(1^\lambda) \rightarrow (pp_{GC}) \text{ takes in the security parameter \( \lambda \) and outputs public parameters \( pp_{GC} \).}
- \textbf{gc.Garble}(pp_{GC}, C_G, d_{in}) \rightarrow (l, T_{1w}, T_{1w}^G, T_{dil}) \text{ takes as input \( pp_{GC} \), a boolean circuit \( C_G \), the input bit string \( d_{in} \), and randomly samples \( r_G, l \in \mathbb{Z}_m \). The output consists of labels \( l \), the undistorted and garbled label-to-table tables \( T_{1w}, T_{1w}^G \), and the bit-to-label table \( T_{dil} \).}
- \textbf{gc.Evaluate}(pp_{GC}, l_{in}^{p_1}, l_{in}^{p_2}, T_{1w}^{G_1}, T_{1w}^{G_2}, T_{out}^{dec}) \rightarrow (d_{out}) \text{ takes in \( pp_{GC} \), input labels \( l_{in}^{p_1}, l_{in}^{p_2} \), the garbled label-to-table table \( T_{1w}^G \), a label-to-bit decoding \( T_{dec}^{out} \), and outputs the bit string \( d_{out} \).}

On a high-level, a 2PC system based on boolean garbled circuits involves a party \( p_1 \) as the garbler and party \( p_2 \) as the evaluator. Party \( p_1 \) calls \textbf{gc.Setup} and \textbf{gc.Garble}, where \( p_1 \) loops over every wire of the circuit \( C_G \) 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 \( T_{dil} \) and \( T_{1w} \) provide a bit-to-wire mapping of all possible bit combinations. Further, the wires \( w \) and with that table \( T_{1w} \) encode the computation logic of \( C_G \). Next, \( p_1 \) uses the randomness \( r_G \) to turn \( T_{1w}^G \) into a garbled representation \( T_{1w}^G \), and determines \( l_{in}^{p_1} \) at input wires according to input bits \( d_{in}^{p_1} \). Subsequently, \( p_1 \) sends \( l_{in}^{p_1}, T_{1w}^{dec}, T_{1w}^G \) to \( p_2 \).

Next, to obtain the remaining input labels \( l_{in}^{p_2} \), \( p_1 \) and \( p_2 \) interact with the OT\_2 scheme defined in Section A5c. Because, \( p_1 \) is not allowed to learn the private input \( d_{in}^{p_2} \) of \( p_2 \). Hence, per input bit of the string \( d_{in}^{p_2} \), \( p_1 \) sends labels encrypted by ot.Encrypt as messages \( (m_1=l_{in}^{d_{in}^{p_2}} \text{ and } m_2=l_{in}^{d_{in}^{p_2}}) \) to \( p_2 \). Party \( p_2 \) receives labels \( l_{in}^{d_{in}^{p_2}} = m_2 \) by calling ot.Decrypt and, with that, \( p_1 \) does not learn any bits \( d_{in}^{p_2} \). With access to the labels \( l_{in}^{d_{in}^{p_2}}, l_{in}^{d_{in}^{p_2}}, p_2 \) is able to evaluate \( T_{1w}^G \) inside \textbf{gc.Evaluate} and, with the label-to-bit decoding table \( T_{dec}^{out} \), \( p_2 \) translates computed output labels back to the bit string \( d_{out} \).

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

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

Commitment schemes are often used in protocols which rely on ZKP cryptography. Using a ZKP to compute the \textbf{c.Open} function allows a prover to convince the verifier from knowing a valid commitment opening without revealing the witness.

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

- \textbf{zk.Setup}(1^\lambda, C) \rightarrow (CRS_C) \text{ takes in a security parameter and algorithm, and yields a common reference string,}
- \textbf{zk.Prove}(CRS_C, w, x) \rightarrow (\pi) \text{ consumes the CRS, public input } x, \text{ and the private witness } w \text{ and outputs a proof } \pi.
- \textbf{zk.Verify}(CRS_C, x, \pi) \rightarrow \{0, 1\} \text{ yields true (1) or false (0) upon verifying the proof } \pi \text{ 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 \textbf{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 \textbf{zk.Setup} function can be called by either party. The function \textbf{zk.Prove} and \textbf{zk.Verify} are called by the prover and verifier respectively.

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

B. Security Analysis

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.

1) 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.

a) Session Confidentiality: In the handshake phase, semi-honest 2PC circuits are executed as follows. Before evaluating the circuit \( C_{SHTS} \) with labels that output the secret SHTS, the \textit{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 \( C_{SHTS} \) for CHTS happens after the exchange of SHTS. Notice that CHTS is never shared with the \textit{proxy}. Further, if the \textit{proxy} honestly garbles \( C_{SHTS} \) and passes server-side messages back to the \textit{client}, then the SHTS verification at the \textit{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]
TLS 1.3-specific computations achieves session-confidentiality against a probabilistic polynomial time adversary $A$ which maliciously garbles $C_{XHTS}$, 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 $C_{XHTS}$ 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 $C_{SHTS}$ 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 B2a), or lets clients detect malicious proxies after the disclosure of TLS session secrets (cf. Appendix B2b).

**CHTS:** The semi-honest 2PC evaluation of the circuit $C_{CHTS}$ is secure under the same cryptographic schemes as the evaluation of the circuit $C_{SHTS}$. 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 $C_{CHTS}$ 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 $C_{CHTS}$ preserves session-confidentiality, because the adversary remains with a negligible probability of guessing $s_2$.

b) **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 $A$ 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 $C_{XHTS}$ 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.

c) **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 $A$ 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.

**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

$^5$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.
and obtain authenticated message transcripts which depend on a secret server randomness.

d) 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, the semi-honest 2PC evaluation and disclosure of SHTS achieves authenticity for the proxy against a probabilistic polynomial time adversary 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.

2) 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.

a) 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 A, which maliciously garbles 2PC circuits, acts as a malicious verifier, or deviates as the evaluator in 2PC systems.

**Proof 5.** For the adversary A₁, acting as a client, the interaction to compute all record and post-record circuits C(k,iv), CECB, CTag, Copen, and Cskopen never leak secret information of the proxy due to the secure OT and garbling scheme. Thus, A₁ never learns secret shares of the proxy. For an adversary A₂, acting as the proxy, no 2PC interactions to compute the circuits C(k,iv), CECB, CTag leaks a bit due to the deployment of the maliciously secure 2PC system in the record phase. In the post record phase, the adversary A₂ does not learn any bit throughout the semi-honest interaction in the proof system to compute Cskopen, 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 A₂. 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 Copen, session-confidentiality is not preserved anymore because A₂ learns the data which is transparently shared.

b) 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 A which acts as a cheating prover, a malicious garbler, or injects arbitrary TLS parameters or data throughout the protocol.

**Proof 6.** The adversary A₁, acting as a client, cannot inject arbitrary response records into the Janus protocol because the adversary A₁ 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 (x, t²), which has been selected by the adversary, in the semi-honest proof system, the adversary A₁ cannot inject arbitrary response data into the Janus protocol. The adversary A₁ 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 $A_1$ for the circuits $C_{tpOpen}$ and $C_{zkOpen}$ is caught, because the honest proxy verifies input against an already authenticated SHTS parameter (cf. Section B1d). 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 $A_2$, taking the role of a malicious proxy, maliciously garbles the 2PC circuits $C_{(k_{m}, iv)}$, $C_{ECB_2}$, and $C_{tag}$, 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 $C_{tpOpen}$ and $C_{zkOpen}$ is detected by the client after the proxy discloses the garbled truth tables in the commit-disclose-open paradigm. Further, we consider the adversary $A_2$ 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' = 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 $A_2$ 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 $C_{tpOpen}$ and $C_{zkOpen}$, only the client inputs private data such that the adversary $A_2$ can only deviate form the Janus protocol if $A_2$ maliciously garbles the circuits. As already stated in the previous sentences, a malicious garbling of $A_2$ in the record and post-record phase is infeasible.