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*—TLS oracles guard the transition of web data from an authenticated session between a client and a server to a data representation that any third party can verify. Current TLS oracles resolve weak security assumptions with cryptographic algorithms that provide strong security guarantees (e.g., maliciously secure two-party computation). However, we notice that the conditions and characteristics of TLS 1.3 allow for reconsidering security assumptions. Our work shows that the deployment of semi-honest two-party computation is feasible with a single exception, while retaining equivalent security properties. Further, we introduce a new parity checksum construction to decouple the integrity verification over AEAD stream ciphers into dedicated proof systems and improve end-to-end performance benchmarks. We achieve a selective and privacy-preserving data opening on 16 kB of TLS 1.3 data in 2.11 seconds and open 10x more data compared to related approaches. Thus, our work sets new boundaries for privacy-preserving TLS 1.3 data proofs.**

***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 data provenance. 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 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*, which achieves 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 such that data provenance becomes publicly verifiable.

TLS-oracles have originated in the context of blockchain ecosystems, where TLS-oracles originally solved the "oracle problem" of trustworthy data imports with a cryptographically verifiable origin and correctness. TLS-oracles, however, are generally applicable in the Internet, which makes them a crucial technique to build user-centric and data-sovereign systems [5]. For instance, through TLS-oracles, users are able to present solvency checks without giving up control of their data [6]. As of today, multiple deployment types of TLS-oracles exist, even TLS-oracles that support private data proofs. However, the efficiency of privacy-preserving TLS-oracles is limited depending on the employed approach. As a result, privacy-preserving TLS-oracles are not applicable to, among others, business and legal domains, where guaranteed delivery of larger data objects (e.g., confidential files) is necessary (e.g., to build verifiable accountability [7]). Because current methods are unsatisfactory, we set out to improve the scalability of TLS 1.3 data provenance.

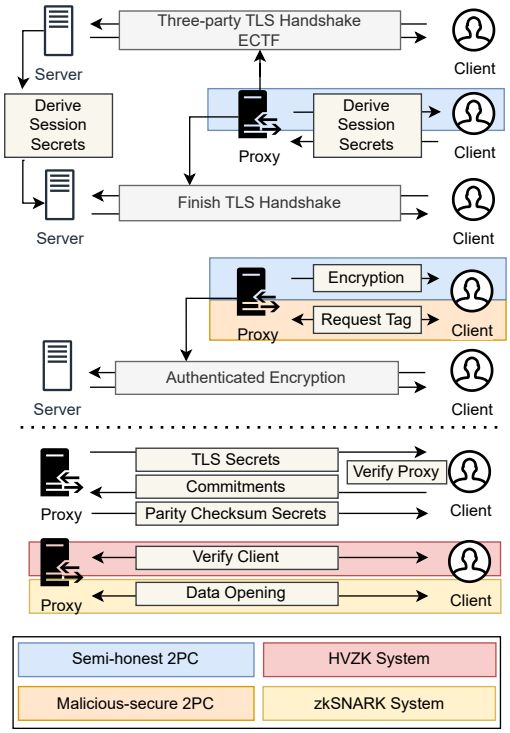

Fig. 1. Overview of the *Janus* protocol, where the *client* and a *proxy* collaboratively run a TLS session with the *server*. At the end of the TLS session (dotted horizontal line), the *client* continues to interact with the *proxy* in order to convince the *proxy* of a valid TLS data opening.

In the context of TLS-oracles, the TLS client side is a collaborative and secret-shared session between two parties: the trusted verifier and the *client*. The mutual client-side ses-

sion guarantees that the *proxy*, without access to confidential parameters of the TLS session, can audit if the *client* preserves session integrity according to the TLS specification. In this work, we expect the trusted verifier to act as a *proxy* between the *client* and the *server*. In contrast to other works [4], we assume that malicious *clients* can mount machine-in-the-middle (MITM) attacks between the *proxy* and the *server*. TLS-oracles require the mutually shared TLS session between the *proxy* and the *client* to process messages using maliciously secure 2PC [3], [4], [8]. This way, the security of secret-shared TLS parameters is preserved such that the *proxy* maintains the control to audit session integrity of the *client*.

In the presented setting, and as the first contribution, our work makes the key observation that in the TLS 1.3 handshake phase, picking and setting the *client* key share in the client hello (CH) message can yield an immediate key derivation at the *server*. With that, the *server* immediately responds to both client-side parties with authenticated messages. In our system model, which prevents forging of *server* certificates, we observe that the adversary model in the TLS 1.3 handshake phase can be reduced to a setting where the *client* and *proxy* interact using semi-honest 2PC. To proceed according to the TLS specification when using semi-honest 2PC, both client-side parties eventually validate 2PC inputs or outputs against the authenticated handshake parameters which have been initially verified (e.g. server handshake traffic secret (SHTS)) with the help of authenticated server-side messages. As such, all computations according to the TLS 1.3 specification can be reduced to a semi-honest 2PC setting except for the computation of request authentication tags (cf. Figure 1).

Our second contribution concerns the post-processing phase of TLS-oracles, where the *client* opens data from the shared TLS session towards the *proxy*. The *proxy* attests to opened data if the integrity on opened data is *correct* against the audited and *authentic* TLS 1.3 session. If TLS-oracles support privacy, then the *client* opens data by presenting a privacy-preserving zero-knowledge proof (ZKP) to the *proxy*. The ZKP ensures that the *proxy* learns nothing beyond the validity of a statement that holds on the opened data. Initial zero-knowledge based openings of TLS data computed on small data sizes (e.g., proving JSON key value pairs) [4]. Because, to compute a data opening in a ZKP circuit, the *client* proves that an *authentic* ciphertext, which is intercepted at the *proxy*, has a non-ambiguous mapping to a plaintext. To prove the mapping, the ZKP circuit computes legacy algorithms with non-algebraic structures (e.g., AES128), which are inefficient to evaluate in current zkSNARK proof systems [9]. In TLS 1.3, privacy-preserving data openings benefit from the structure of authenticated encryption with associated data (AEAD) stream ciphers that protect exchanged TLS traffic. Here, current works reduce the complexity of the ciphertext-to-plaintext mapping to an efficient XOR computation which relies on an authenticated stream of pseudorandom parameters. Nevertheless, the optimization comes at a cost. In the works [3], [8], the *client* must know how data is organized in order to selectively open the data of interest. Without knowledge of how TLS data is structured, the opening complexity depends on Merkle Tree inclusion proofs. The work [9] partly resolves the overhead introduced by legacy algorithms in zkSNARK proof systems by decoupling the computation of legacy algorithms into a record independent and record dependent computation phase.

Our work, by contrast, leverages our first contribution and moves the computation structures of legacy algorithms into a honest verifier zero-knowledge (HVZK) proof system, which is based on a semi-honest 2PC setting [10]. The HVZK proof system is able to efficiently evaluate non-algebraic structures and, as such, efficiently verify legacy algorithms. In the first step, we authenticate pseudorandom parameters of TLS 1.3 stream ciphers via a new secret parity checksum computation inside the HVZK proof system (cf. red box in Figure 1). In the second step, we efficiently open private TLS data with the help of the parity checksum inside a zkSNARK proof system (cf. yellow box in Figure 1). With our protocol, we achieve new scales of privacy-preserving data openings in the context of TLS 1.3 and open 16 kB of private TLS 1.3 data in 2.11 seconds, which outperforms related approaches by a factor of 10x (cf. Section VI). Our protocol allows selective privacy-preserving or transparent data openings and maintains a low communication overhead throughout the TLS 1.3 handshake and record phase.

In analogy to Roman mythology, we name our efficient oracle solution after the god of transitions, Janus. Because, with our protocol, TLS 1.3 oracles in the proxy setting transition into an area with new boundaries concerning large-scale and privacy-preserving data proofs. We contribute to the state of the art of data provenance protocols as follows:

- We introduce *Janus*, a novel protocol for data provenance via TLS 1.3, by optimizing secure computation structures based on characteristics found in TLS 1.3. *Janus* retains security properties equivalent to previous works and sets new benchmarks for proving kilobytes of data.
- We show that TLS 1.3 SHTS authenticity can be verified at both client parties in a semi-honest 2PC setting. With access to an authentic SHTS parameter, both client parties can mutually verify the authenticity of subsequent semi-honest 2PC interactions except for the computation of request authentication tags (cf. Section IV).
- We improve the data scalability of selective and privacy-preserving data proofs in TLS 1.3 by having the *proxy* verify the TLS integrity in dedicated zero-knowledge proof systems (cf. Section V-D). The decoupling of the TLS integrity verification relies on a new masked parity checksum which ensures that both proof systems evaluate data that has been derived from authenticated and consistent TLS session parameters.
- We analyse the security of *Janus* (cf. Appendix B), provide end-to-end evaluation benchmarks (cf. Section VI), and open-source[1] the secure computation building blocks.

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

TABLE I
NOTATIONS AND FORMULAS OF TLS VARIABLES.

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

II. PRELIMINARIES ON TLS ORACLES

This section introduces where and how TLS-oracles use cryptographic building blocks to modify the TLS 1.3 baseline such that the provenance and correctness of TLS session data can be publicly verified. Througout this section, we explain the main functionalities of cryptographic building blocks and provide further details of each cryptographic construction or protocol in the Appendix A.

A. General Notations

The TLS notations of this work are introduced in Section II-B, and closely follow the notations of the work [11]. Further, we denote vectors as bold characters $\mathbf{x} = [x_1, \ldots, x_n]$, where $len(\mathbf{x}) = n$ returns the length of the vector. Base points of elliptic curves are represented by $G \in EC(\mathbb{F}_p)$, where the finite field $\mathbb{F}$ has a prime size p. For elliptic curve elements, the operators $\cdot, +$ refer to the scalar multiplication and addition of elliptic curve points $P \in EC(\mathbb{F}_p)$. The symbol λ indicates the security parameter. For bits or bit strings, the operators $\cdot, \oplus$ represent the logical AND or multiplication, and the logical XOR or addition respectively. Other operators describe a random assignment of a variable with $\xleftarrow{\$}$, the concatenation of strings with $||$, and the comparison of variables with $\stackrel{?}{=}$.

B. Transport Layer Security

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

TLS Handshake between the client c and server s:

inputs: $x \xleftarrow{\$} \mathbb{F}_p$ by c. ($y \xleftarrow{\$} \mathbb{F}_p$, sk_S, pk_S) by s.
outputs: $(\text{tk}_{\text{CATS}}, \text{iv}_{\text{CATS}}, \text{tk}_{\text{SATS}}, \text{iv}_{\text{SATS}})$ to c and s.

1. c: $X = x \cdot G$; send X in m_{CH}
2. s: $Y = y \cdot G$; send Y in m_{SH}
3. b: dES = **hkdf.exp**(**hkdf.ext**(0,0),"derived" || H(" "))
4. b: DHE $= x \cdot y \cdot G$; HS = **hkdf.ext**(dES, DHE)
5. b: SHTS = **hkdf.exp**(HS,"s hs traffic" || H_2)
6. b: CHTS = **hkdf.exp**(HS,"c hs traffic" || H_2)
7. b: $(k_{\text{CHTS}}, iv_{\text{CHTS}})$ = DeriveTK(CHTS)
8. b: $(k_{\text{SHTS}}, iv_{\text{SHTS}})$ = DeriveTK(SHTS)

9. b: fk_S = **hkdf.exp**(SHTS, "finished" || " ")
10. s: SCV=**ds.Sign**(sk_S,label_{11}||H_6); send SCV in m_{SCV}
11. s: SF = **hmac**(fk_S, H_7); send SF in m_{SF}
12. c: SF' = **hmac**(fk_S, H_7); verify SF' $\stackrel{?}{=}$ SF
13. c: **ds.Verify**(pk_S, label_{11} || H_6, SCV) $\stackrel{?}{=} 1$
14. b: fk_C = **hkdf.exp**(CHTS, "finished" || " ")
15. c: CF = **hmac**(fk_C, H_9); send CF in m_{CF}
16. s: CF' = **hmac**(fk_C, H_9); verify CF' $\stackrel{?}{=}$ CF

17. b: dHS = **hkdf.exp**(HS,"derived" || H(" "))
18. b: MS = **hkdf.ext**(dHS, 0)
19. b: CATS = **hkdf.exp**(MS, "c ap traffic" || H_3)
20. b: SATS = **hkdf.exp**(MS, "s ap traffic" || H_3)
21. b: $(k_{\text{CATS}}, iv_{\text{CATS}})$ = DeriveTK(CATS)
22. b: $(k_{\text{SATS}}, iv_{\text{SATS}})$ = DeriveTK(SATS)

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

1) Handshake Phase: To establish a secure channel between a server and a client, TLS 1.3 relies on the Diffie-Hellman key exchange (DHKE) to securely exchange cryptographic secrets between two parties (cf. Figure 2, lines 1-4). With TLS configured to use elliptic curve cryptography, parties protect secrets x, y with an encrypted representation X, Y. Next, the values X, Y are exchanged in plain via the CH and server hello (SH) messages m_{CH}, m_{SH}. With access to X, Y, both parties derive the Diffie–Hellman ephemeral (DHE) key, where DHE $= x \cdot y \cdot G = y \cdot X = x \cdot Y$ holds. Both parties use the DHE value to compute the handshake secret (HS), and, with that, derive the SHTS and client handshake traffic secret (CHTS) (cf. Figure 2, lines 5,6). SHTS and CHTS are used to derive client-side and server-side handshake keys and initialization vectors (cf. Figure 2, lines 7,8), which secure all subsequent handshake messages.

To mutually authenticate each other, both parties exchange certificates and compute authentication parameters (cf. Figure 2, lines 9-16). Notice that in TLS, client-side authentication is optional, which is why we omit client certificates in Figure 2. But, we show the computations of the server finished (SF) and client finished (CF) authentication values, because, to constitute a valid TLS session, both parties must successfully exchange and verify the SF and CF messages

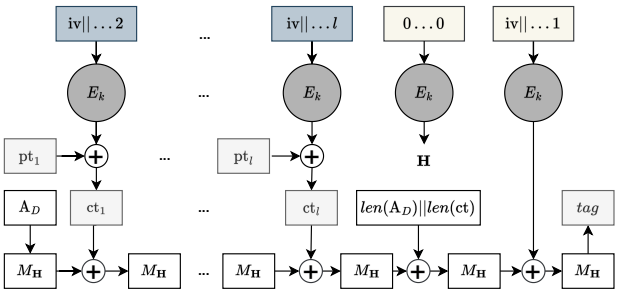

Fig. 3. AEAD stream cipher configured with AES in the Galois/Counter mode (GCM). The algorithm encrypts a plaintext $\mathbf{pt} = [pt_1, \ldots, pt_l]$ to a ciphertext $\mathbf{ct} = [ct_1, \ldots, ct_l]$ under key k and authenticates the ciphertext **ct** and associated data A_D with the tag t. The symbol $M_\mathbf{H}$ is a Galois field multiplication which translates bit strings into GF(2^{128}) polynomials, multiplies the polynomials modulo the field size and translates the polynomial back to the bit string representation.

$m_{\text{SF}}, m_{\text{CF}}$. For server-side authentication, the server computes the server certificate verify (SCV) value, which binds a Public Key Infrastructure (PKI) X.509 certificate to the TLS 1.3 transcript via a digital signature [12]. Here, the signature is computed with the server secret key sk_S and is verified with the corresponding server public key pk_S. The client obtains the server public key pk_S in the PKI certificate and aborts the TLS session if the signature verification fails.

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

2) Record Phase: The TLS record phase requires parties to protect data with an AEAD algorithm before data can be exchanged. The AEAD algorithm in TLS 1.3 depends on the previously established application traffic keys and initialization vectors and translates plaintext data **pt** into a confidential and authenticated representation $(\mathbf{ct}, t)$, with ciphertext **ct** and authentication tag t. 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.

The record phase of TLS 1.3 allows three configurations of AEAD stream ciphers to protect exchanged data between the client and the server. Stream ciphers are characterized by pseudorandom generators, which incrementally output key streams or encrypted counter blocks (ECBs). In a next stage, ECBs are combined with plaintext data chunks to compute ciphertext data chunks. Subsequently, AEAD ciphers compute an authenticated tag on all ciphertext chunks and associated data. For instance, the TLS 1.3 stream cipher based on the 128-bit Advanced Encryption Standard (AES) in the Galois mode implements the pseudorandom generator with the AES encryption function E_k and authenticates data with a Galois authentication tag t (cf. Figure 3). We elaborate on the AEAD algorithm in the Appendix A4

C. 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 clients eventually present (cf. Figure 1). If the verification succeeds, the third-party 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) 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 integrity of TLS data, the verifier and client join a mutually vetting but collaborative TLS client session. To construct a joint client-side TLS session, TLS-oracles replace the TLS handshake with a three-party handshake (3PHS). In the 3PHS, every party injects a secret randomness such that the DHE secret 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.

The consequence of the 3PHS is that the client depends on the computational interaction with the 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 TLS secrets and encryption keys that are required for the secure session with the server. And, the introduction of false session data at the client leads to a session abort at the server.

2) Client-side Two-party Computation: With mutually dependent TLS secret, 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, where the first efficient 2PC approach used in TLS-oracles is secure computation based on boolean Garbled Circuits (GCs) [13]–[15]. Computations in boolean GCs is efficient if the 2PC function can be efficiently expressed in boolean logic. With TLS 1.3, suitable algorithms for boolean circuit computations exist. For instance, the hash functions of the key derivation algorithms (e.g. Secure Hash Algorithm (SHA) with 90,825 AND gates), or the record phase encryption functions (e.g. AES with 6,800 AND gates) have optimized binary circuit representations [16]. However, after the 3PHS, parties are in control of arithmetic secret shares and boolean garbled circuits are inefficient in expressing arithmetic computations. For instance, the arithmetic operations for an addition of the elliptic curve (EC) point x-coordinate requires four subtractions, a modular inversion, and two modular multiplications, which leads to an estimated circuit complexity with 900,000 AND gates [4],

[17]. Atop, arithmetic garbled circuits are neither a choice because field elements depend on garbled truth tables that scale quadratically with respect to the selected field size [18].

To remain efficient, TLS-oracles translate the arithmetic addition operation of the EC x-coordinate shares into a bitwise additive operation, which can be efficiently expressed in a boolean circuit. The additive operation is of interest, as 2PC circuits must reconstruct TLS parameters before a key derivation or encryption circuit can be called. The 2PC technique to translate additive arithmetic shares into bitwise additive shares is the Elliptic Curve to Field (ECTF) protocol [3], [4]. After running the ECTF protocol, the verifier and the client obtain secrets with a bitwise sum that match the EC x-coordinate of the DHE secret. Further details of the ECTF conversion and secure 2PC techniques are provided in the Appendix A5.

3) Public Verifiability of TLS Data: The last phase of TLS-oracles decouples the client-side computation dependence and provides the client with public verifiability of TLS data. In the initial TLS-oracle approache [2], [3], the decoupling concerns the 2PC garbled circuit $\mathcal{C}_G^{\text{ECB}}$ which computes record phase ECBs. Here, the client acts as the circuit evaluator, obtains a labelled output encodings, and shares a commitment of ECB encodings to the verifier. After receiving the commitment, the verifier discloses encryption secret shares to the client by revealing the bit-to-label decoding tables of the $\mathcal{C}_G^{\text{ECB}}$ inputs and outputs. Further, the verifier attests to TLS data by signing the commitments, the output bit-to-label decoding table, and the transcript hash of the ciphertext in a certificate, which is shared with the client. The client is now able to verify the correctness of $\mathcal{C}_G^{\text{ECB}}$ labelled output encodings, decode ECBs, and with that, decrypt the record phase ciphertext. In the last stage, clients can selectively present a tuple of labelled output encordings, ciphertext chunks, the output decoding table, and the certificate with any third-party. Every third-party, who trusts the TLS-oracle verifier, is able to publicly verify the integrity and provenance of the presented TLS data.

Since the client is the only party with access to labelled output encordings, clients can optionally prove TLS data by using a privacy-preserving ZKP. In this case, the ZKP circuit (i) takes in private labelled output encordings, (ii) computes the ECBs with the help of the public decoding table, and (iii) authenticates TLS data by showing that a plaintext xored with ECBs yield a ciphertext, which matches the public ciphertext transcript hash. Last, the circuit shows that labelled output encordings match their signed public commitment and that the authenticated plaintext complies with a given data policy.

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 & Threat Model

- ***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.

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 zero-knowledge or HVZK hold.

B. System Goals

The *Janus* protocol supports 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* 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 of the TLS handshake 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.

- ***MITM-resistance*** guarantees that even if the *client* is able to intercept traffic between the *proxy* and the *server*, the malicious *client* cannot convince a *proxy* of data proofs on modified server-side traffic.
- ***Legacy-compatibility*** holds if the default TLS code stack running at the *server* does not require any changes and achieves out-of-the-box compatibility with our protocol.
- ***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.
- ***Data-provenance*** holds if the *client* proves a verifiable and deterministic computation trace on presented TLS data by leveraging confidential TLS session parameters with *session-integrity* and *session-authenticity*.

IV. TLS 1.3 Observations

Current TLS-oracles with *MITM-resistance* retain confidentiality and integrity in the presence of malicious parties by employing malicious secure 2PC [2], [4], [8]. In this work, we reduce the overhead introduced by maliciously secure 2PC and show that partly employing semi-honest 2PC is possible. With semi-honest 2PC, our protocol achieves confidentiality of TLS session secrets against the *proxy* as the verifier and prevents a malicious *client* from deviating the TLS 1.3 specification. We expect the *proxy* to act as the garbler in semi-honest 2PC computations. The deployment of the semi-honest 2PC system is possible due to the following important characteristics and conditions of TLS 1.3.

A. Authenticity for SHTS

Our initial observation concerns the 2PC circuits which compute and disclose the SHTS and the CHTS secrets. Notice that due to the key independence property of TLS 1.3, disclosure of handshake traffic secrets does not compromise the security of TLS 1.3 application traffic keys [19]. This means that if the 2PC circuit correctly outputs SHTS or CHTS, neither the *client* nor the *proxy* can learn anything about the secret HS, which would leads parties to application traffic keys. If the semi-honest 2PC circuit of SHTS discloses the output to the *client*, then the *client* is able to verify correct garbling and input provision of the *proxy* by validating SHTS against the SF message. One key observation is that in TLS 1.3, it is possible to have the *client* pick a supported cipher suite and add keying material in the CH message, such that the *server* directly derives traffic secrets and returns authenticated messages. Thus, if a policy of our TLS-oracle initially determines a cipher suite, then the *server* is expected to respond with authenticated messages after obtaining the CH message. A succeeding verification of authenticated messages based on SHTS implies that a correct SHTS circuit has been garbled by the *proxy*. If the verification fails, then the *client* aborts the protocol. Otherwise, the *client* shares SHTS with the verifier, such that the verifier is able to validate TLS session authenticity by decryption the server-side certificate with SHTS. If the *client* inputs incorrect secret shares into the 2PC computation of CHTS, then incorrectly derived authentication tags lead to session aborts at the *server*.

Malicious garbling of the CHTS circuit does not yield any information leakage to the *proxy* either. Because, if an incorrectly garbled CHTS circuit outputs the DHE secret share of the *client*, then the *client* continues to compute client handshake traffic keys based on a wrong CHTS secret. Even if incorrectly computed client handshake traffic keys protect client-side handshake messages with digests of a ciphertext and authentication tag, then a *proxy* cannot learn any data of the client secret share by viewing the ciphertext and authentication tag.

B. 2PC Authenticity via SHTS

In our work, the two-party computations of the record phase follow the garble-then-prove paradigm of the work [20], where the garble phase comprises 2PC circuits that (i) drive application traffic secrets and (ii) compute AEAD parameters for the construction of application traffic records. In the garble phase, all 2PC circuits can be computed in a semi-honest setting except request authentication tags. Request authentication tags must be computed in a maliciously secure 2PC system because the authentication tag is made public as a part of the request record. If the *proxy* maliciously garbles the authentication tag circuit such that the circuit outputs *client* secrets, then the *proxy* is able to access *client* secret shares via the false authentication tag of the request record. As a result, *session-confidentiality* of client secrets is compromised. Further, the maliciously secure computation of the request authentication tag prevents a malicious *client* from mounting a MITM attack. Because if *clients* cannot compute valid authentication tags individually, *clients* cannot compile another valid request that passes the AEAD verification at the *server*.

Another key observation of our work is that, in TLS 1.3, the prove phase of the garble-then-prove paradigm can be computed with the HVZK system of the work [10]. Due to the fact that TLS 1.3 allows both the *proxy* and *client* to initially validate authenticity via the SHTS parameter (cf. Section IV-A), the HVZK circuit can use the SHTS parameter to verify input correctness. To verify input correctness of secret shares, the HVZK circuit derives and matches a SHTS' against the authenticated SHTS value. Further, the deployment of the HVZK proof system is feasible because in the case of a malicious garbling, the *proxy* learns at maximum a single bit and the *client* is supposed to abort the protocol if the 2PC output of the HVZK circuit yields multiple bit. Since our protocol ensures that the parameters computed in the HVZK circuit are of sufficient randomness, leaking a single bit is bearable [21]. For record data proofs, our protocol makes use of zkSNARK proof systems which do not leak any bits (cf. Section V-F).

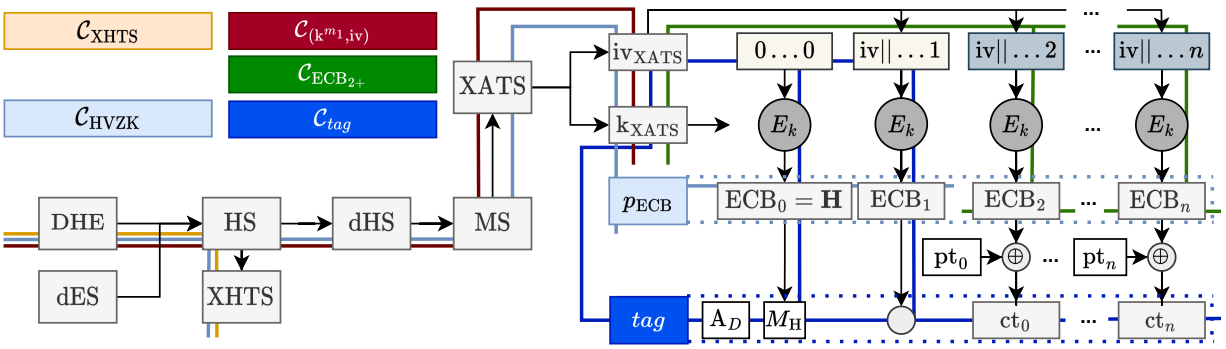

Fig. 4. Overview of 2PC circuits of the *Janus* protocol. All circuits are executed in a semi-honest adversary setting except the circuit $\mathcal{C}_{tag}$, which executes with security against malicious adversaries. Depending on provided inputs, the circuits output server-side or client-side TLS 1.3 parameters.

V. Janus: Fast Privacy-preserving Proxy Oracle for TLS 1.3

The following sections provide the details of the *Janus* protocol (cf. Figure 1). The subsection V-B summarizes all required 2PC circuits and introduces our construction of a masked parity checksum. Subsequent subsections describe the *Janus* protocol specification which ends with two different TLS data attestation modes.

A. System Setup

Our protocol builds upon system roles introduced in the Section III-A, where the *proxy* takes the role of the TLS-oracle verifier. As such, the *proxy* acts as the garbler of all semi-honest 2PC circuits as well as semi-honest 2PC circuits for the computation of HVZK proofs. The *client* acts as the evaluator of all 2PC circuits. In employed zkSNARK systems, the *client* takes the role of the prover and the *proxy* acts as the verifer. Initially, the *client* and *proxy* agree on a policy P, which indicates a public statement ϕ that is proven by TLS data that has been queried by the *client*. Next, the *proxy* and *client* execute 2PC and zkSNARK offline computations to set up the required parameters for the 2PC and zkSNARK online computations. 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 in a semi-honest 2PC interaction to collaboratively but vetted 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 4), which, if in the case of semi-honest 2PC, disclose outputs to the *client*.

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

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

Subsequently, for every request, the *proxy* and *client* compute the circuit $\mathcal{C}_{\text{ECB}_{2+}}$, which outputs encrypted counter blocks $\mathbf{ECB}_{2+}$ to the *client*. The notation $\mathbf{ECB}_{2+}$ expresses encrypted counter blocks ECB_i with an index $i > 1$. With access to $\mathbf{ECB}_{2+}$ the *client* computes the ciphertext chunks $\mathbf{q}$ of the request record and shares $\mathbf{q}$ with the *proxy*. Once the *proxy* receives $\mathbf{q}$, both the *proxy* and *client* interact in a maliciously secure 2PC computation of the circuit $\mathcal{C}_{tag}$ (cf. dark blue circuit in Figure 4) to complete the construction of the request. The circuit derives the encrypted counter blocks $\mathbf{ECB}_{tag}$=$[\text{ECB}_0, \text{ECB}_1]$, and, together with the cipher-text chucks $\mathbf{q}$, computes and outputs the request authentication tag t. We follow the bandwidth-efficient 2PC dual-execution paradigm of the works [3], [14] to implement a malicious secure 2PC evaluation of the circuit $\mathcal{C}_{tag}$. The maliciously secure evaluation of $\mathcal{C}_{tag}$ prevents the *client* from accessing $\mathbf{ECB}_{tag}$ values and other secret parameters of the *proxy*. If *clients* access $\mathbf{ECB}_{tag}$ values, non-ambiguity of request ciphertext chunks is broken as *clients* are able to generate compliant TLS session requests. If MITM attacks are feasible and non-ambiguity of request holds, a *proxy* looses the ability to verify *session-integrity* because *clients* may present any

ciphertext to the *proxy* but send a different ciphertext to the *server*. On the other hand, the malicious secure circuit $\mathcal{C}_{tag}$ prevents the *proxy* from changing the circuit output to a TLS session secret share of the *client*, which would be disclosed to the *proxy* as part of the request.

3) Record Response Circuits: After compiling and sending requests to the *server* using 2PC, the *proxy* records (i) request ciphertext and authentication tag pairs ($\mathbf{q}$, t^{q}) and (ii) proceeding response ciphertext and authentication tag pairs ($\mathbf{r}$, t^{r}) from the *server*. When processing server-side response data, the execution of the 2PC circuit $\mathcal{C}_{\mathrm{ECB}_{2+}}$ depends on whether or not there exists a dependency between follow up requests and obtained responses.

a) Requests are independent of responses.: If no request depends on the contents of a response, then the circuit $\mathcal{C}_{\mathrm{ECB}_{2+}}$ is only called for the compilation of request ciphertexts. In Section V-C, we explain that response ECBs can be locally computed by the *client* once proxies disclose session secrets.

b) Requests depend on responses.: If a request with number $n > 1$ depends on the contents of responses $\mathbf{r}$=$[r_1, \ldots, r_l]$, where each response r_m has an index $m < n$, then the *client* and *proxy* perform l executions of the circuit $\mathcal{C}_{\mathrm{ECB}_{2+}}$. The evaluation of l circuits $\mathcal{C}_{\mathrm{ECB}_{2+}}$ yields l vectors of encrypted counter blocks $\mathbf{ECB}_{2+}$ to the *client*. With l vectors of $\mathbf{ECB}_{2+}$, the *client* is capable of accessing the contents of the responses $\mathbf{r}$=$[r_1, \ldots, r_l]$ to construct the n-th request. Further, with access to the l^{th} $\mathbf{ECB}_{2+}$, the *client* is incapable of accessing any TLS session secret share of the *proxy* and, thus, cannot compute any y^{th} $\mathbf{ECB}_{2+}$, with $y>l$. To preserve *MITM-resistance*, it must hold that the *proxy* intercepts the pair $(\mathbf{r}, \mathrm{t}^{\mathrm{r}})$ before the circuit $\mathcal{C}_{\mathrm{ECB}_{2+}}$ outputs the corresponding $\mathbf{ECB}_{2+}$. Notice that any $\mathbf{ECB}_{2+}$ vector does not include any $\mathbf{ECB}_{tag}$ parameters.

C. Commit & Disclose

As soon as the *client* has exchanged enough record data with the *server* such that the statement ϕ can be satisfied, the *client* sends a notification to the *proxy* such that the *proxy* stops the recording of ciphertext authentication tag pairs.

1) Disclose Phase: The disclose phase sets the basis of the mutual TLS parameter verification between the *proxy* and the *client*. In the eyes of the *proxy*, the collection of request and response pairs ($\mathbf{q}$, t^{q}) and ($\mathbf{r}$, t^{r}) represent commitments of the *client* to the underlying TLS plaintext data. Once the *proxy* has obtained enough traffic pairs, the *proxy* discloses all TLS session secret shares to the *client* and, with that, sacrifices the property of *session-confidentiality*. The *client* on the other hand never discloses locally controlled secret shares. With full access to the TLS 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 encrypted counter blocks $\mathbf{ECB}_{tag}$ to the *proxy* such that the *proxy* is able to verify recorded traffic pairs. A traffic pair ($\mathbf{x}$, $\mathbf{t}^{\mathrm{x}}$) is valid if the encrypted counter blocks $\mathbf{ECB}_{tag}$ and the ciphertext chunks $\mathbf{x}$ compute the intercepted authentication tag t^{x}. Notice that at this stage of the *Janus* protocol, the *client* could forge response records with a MITM attack and present valid encrypted counter blocks $\mathbf{ECB}_{tag}$ to the *proxy* such that the traffic authenticity verification succeeds. However, our protocol prevents this attack at a later stage (cf. Section V-E) by having the *proxy* compare obtained $\mathbf{ECB}_{tag}$ parameters against correctly derived ECBs in a HVZK circuit.

2) Commit Phase: After the disclose phase, the *client* computes and discloses a commitment string s_{ECB} to the *proxy*. The formula of the commitment strings depends on the type of the data opening.

a) Commitment String for a Transparent Data Opening: In the case of a transparent data opening, where opened TLS data is eventually disclosed to the *proxy*, a commitment string s_{ECB}=$H(\mathbf{ECB}_{2+})$ (e.g. with H=SHA256) is a hash of all ECBs that are of interest for the data opening. For a transparent data opening, we require the hash of the commitment string to be of a non-algebraic structure. Because, the *proxy* verifies the hash in the HVZK proof system, which efficiently evaluates algorithms that rely on non-algebraic structures. Similar to the verification of correctly disclosed $\mathbf{ECB}_{tag}$ parameters, the HVZK circuit verifies if the *client* commits to valid ECBs of a TLS session (cf. Section V-E). If the *client* commits to a subset of ECBs, then the *client* attaches a list of indices $\mathbf{I}^{\mathrm{ECB}}$ to the commitment string such that the HVZK circuit can determine which ECBs apply for the verification of the commitment string.

b) Commitment String for a Privacy-preserving Data Opening: In the privacy-preserving opening mode, where only the validity of a statement on the opened data is revealed to the *proxy*, the commitment string s_{ECB}=$h^{\mathbf{ECB}}||t^{\mathbf{ECB}}$ computes as the concatenation of a zkSNARK-friendly hash on ECBs (e.g. $h^{\mathbf{ECB}}$=MiMC($\mathbf{ECB}$) [22]) and the parity checksum $p_{\mathbf{ECB}}$. Both the parity checksum $p_{\mathbf{ECB}}$ and the zkSNARK-friendly hash $h^{\mathbf{ECB}}$ process the same vector of ECBs, where the ECBs are of interest for the privacy-preserving opening. We indicate the computation of the parity checksum on a selection of ECBs with the light blue dotted box in Figure 4. Notice that for both types of the commitment string, the *client* can select request and response ECBs and indicates via the list of indices $\mathbf{I}^{\mathrm{ECB}}$ the ECBs that apply for the opening. At this stage of the *Janus* protocol, the *client* is able to compute the commitment hash $h^{\mathbf{ECB}}$ but cannot entirely compute the parity checksum $p_{\mathbf{ECB}}$. The computation of the parity checksum depends on a secret mask m_p which is sampled at the *proxy* and the *client* obtains the mask m_p upon disclosing the commitment hash $h^{\mathbf{ECB}}$ to the *proxy*. Once the *client* obtains the secret mask m_p, the *client* computes and discloses the parity checksum $p_{\mathbf{ECB}}$ to the *proxy*. The *proxy* verifies the correctness of the parity checksum $p_{\mathbf{ECB}}$ according to the list of indices $\mathbf{I}^{\mathrm{ECB}}$ in the HVZK circuit (cf. Section V-E). Notice that in our protocol, the *client* is able to commit to a false set of ECBs via $h^{\mathbf{ECB}'} = MiMC(\mathbf{ECB}')$ and still provide a valid

parity checksum that passes the verification in the HVZK circuit. However, once the *client* opens private TLS data, the zkSNARK circuit validates the commitment hash together with the parity checksum, such that the input $\mathbf{ECB}'$ succeeds the hash verification but fails the parity checksum validation.

Similar to how the *client* verifies honest behavior of the *proxy* by partly recomputing the TLS transcript, the *proxy* verifies honest behavior of the *client* by imposing an HVZK proof computation on the *client*. Before we continue with the description of the HVZK proof interaction in Section V-E, we introduce the remaining details of the parity checksum computation in the next section.

D. Parity Checksum

The parity checksum is a random linear combination of ECB chunks and random bit strings and computes according to the Formula 1. The formula takes as input a ECB vector $\mathbf{ECB}$=$[\text{ECB}_1, \ldots, \text{ECB}_n]$ and a vector of randomly samples bit strings $\mathbf{m} \xleftarrow{\$} \mathbb{R}^k$. The parameter k is of the size k=$len(m_i)$=$len(\text{ECB}_i)$. The $\cdot$ operator expresses a logical AND computation of two bit strings and the $\bigoplus$ operator aggregates masked bit strings via the logical XOR expression.

$$p_{\mathbf{ECB}} = f_{\bigoplus}(\mathbf{ECB}, \mathbf{m}) = \bigoplus_{i=1}^{n} (\text{ECB}_i \cdot m_i) \tag{1}$$

The construction of the parity checksum gives the *proxy* the opportunity to decouple the TLS integrity verification of the *client*. The idea behind the parity checksum is that, with the parity checksum, the integrity of ECBs can be efficiently verified in a zkSNARK system once the HVZK circuit authenticates the conformance of the parity checksum with respect to the TLS session. Since the employed HVZK proof systems operates on non-algebraic structures [10] and zkSNARK proof systems efficiently computes on algebraic structures [23], [24], the computation of the parity checksum must be compatible with both systems. Further, in the context of the *Janus* protocol, the parity checksum (i) prevents the *proxy* from learning any ECBs and, with that, TLS data in plain, and (ii) prevents the *client* from injecting arbitrary data which passes the TLS-oracle data provenance verification. The parity checksum as defined with the Formula 1 fulfills the efficiency requirements and secures the construction of the commitment string for privacy-preserving data openings as we show in the Appendix B.

E. Honest-Verifier Zero Knowledge Proof

The next interaction between the *proxy* and the *client* is the computation of a HVZK proof. The computation of the HVZK proof ensures (i) that the *client* inputs correct values throughout all 2PC interactions with the *proxy* and (ii) that the *client* releases correct 2PC outputs to the *proxy*. Figure 4 indicates the HVZK circuit in light blue. The circuit validates input correctness of the *client* by matching the private input s_2 against the publicly known secret SHTS. Remember that the correctness and authenticity of the SHTS secret has been determined by the *proxy* by validating the SF message and the *server* certificate in the TLS handshake phase.

In the case of a transparent data opening, the HVZK circuit computes and matches a SHA56 hash on a selection of ECBs against the transparent commitment string of the *client*. In the scenario of privacy-preserving data openings, the HVZK circuit matches derived ECBs against the parity checksum $p_{\mathbf{ECB}}$ to authenticate the commitment string s_{ECB}. As indicated in previous sections, the parity checksum $p_{\mathbf{ECB}}$ requires as public input a list of indices $\mathbf{I}^{\text{ECB}}$ such that the circuit can consider a valid selection of ECBs. The *client* computes and shares the list $\mathbf{I}^{\text{ECB}}$ with the *proxy* after identifying necessary record data which satisfies the policy statement ϕ. Let us consider an example, where TLS is configured to use AES as the encryption function E. If proving TLS data according to the statement ϕ depends on the data inside the third ciphertext chunk of the first response (r_1, t_1^{r}), then the index=3 is included in the list of indices $\mathbf{I}^{\text{ECB}}$. With $\mathbf{I}^{\text{ECB}}$, the HVZK circuit is able to compute the right $\text{ECB}_{2+\text{index}}$, and consider ECB_5=AES$(k, \text{iv}||\ldots 5)$ for the computation of the response commitment string s_{ECB}. Once the *proxy* successfully verifies the HVZK proof, the *client* is able to open plaintext data pt_3 which corresponds to the third ciphertext chunk $ct_3 \in \text{r}_1$ of the response (r_1, t_1^{r}). The opening of the third plaintext chunk succeeds because it holds that (i) the plaintext chunk matches an authenticated ciphertext chunk with $ct_3 = pt_3 \oplus \text{ECB}_5$ and (ii) the ECB_5 together with other ECBs hashes to the commitment h^{ECB}.

Further, for every record (x, t^x) to be opened by the *client*, the HVZK circuit computes and matches the corresponding encrypted counter blocks $\mathbf{ECB}_{tag}$ against previously shared $\mathbf{ECB}_{tag}$ parameters. The $\mathbf{ECB}_{tag}$ verification ensures that records comply with the TLS session. Notice that the verification of $\text{ECB}_0, \text{ECB}_1$ values provides *MITM-resistance* because a *client* cannot initially predict correct $\text{ECB}_0, \text{ECB}_1$ parameters. The HVZK circuit can be optimized due to the fact that hash functions H of TLS cipher suites depend on secure compression functions (e.g. SHACAL-2 if H=SHA256). All optimizations of the HVZK circuit lead to a more complex circuit representation which we present in the Appendix C.

The sequence of the HVZK proof validation requires a commit-and-reveal interaction between the *proxy* and the *client*. The commitment prevents the *proxy* as a malicious garbler of the HVZK circuit to obtain any *client* secrets and gives the *client* the opportunity to verify garbling correctness of the HVZK 2PC circuit. The commit-and-reveal protocol works as follows. The *client* is required to share a commitment with randomness r_c of the HVZK output labels 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 output label corresponding to a one and aborts the protocol if any output label verification fails.

F. Data Attestation Modes

Once the *proxy* successfully verifies the HVZK circuit and the *client* successfully evaluates correctness of all 2PC outputs, our protocol continues with the data attestation phase. The *proxy* as the oracle verifier attests to TLS data of the *client* if the TLS data correctly opens to either one or multiple commitment strings $\mathbf{s}_{\text{ECB}}$. The *Janus* protocol supports different opening modes and allows the *client* to configure commitment openings in the policy P. The opening mode is either *transparent*, such that presented TLS data is disclosed to the *proxy*, or *privacy-preserving*, such that the *proxy* does not learn anything except the validity of TLS data compliance against the public statement ϕ (cf. Figure 5). If the opening succeeds, the *proxy* attests to TLS data of the *client* by certifying the set of parameters p_{cert}=(t, ϕ, pk, σ) with a signature σ=**ds.Sign**(sk,[ϕ, t]) computed at time t. If the *client* presents p_{cert} to any third party who trusts the *proxy*, then the third party is able to verify TLS data provenance. In the following, we outline the different opening modes, which are supported by the *Janus* protocol.

1) Transparent Opening: In the transparent opening mode, the *client* discloses request and response plaintext chunks **pt** by sending the opening data to the *proxy*. The *proxy* checks the validity of the data opening by executing the **tpOpen** algorithm (cf. Figure 5), which successfully evaluates with an output of one. The **tpOpen** algorithm verifies plaintext chunks against previously verified commitments on request and record ECBs. Since the transparent data opening selectively discloses TLS session data publicly, no statement is verified in the **tpOpen** algorithm. But, the *proxy* overwrites the statement $\phi = H(\mathbf{pt})$ to the hash of authenticated data. This way, every third-party is able to verify data authenticity by verifying the *proxy* attestation on ϕ and can individually verify public TLS data of the *client* against any statement.

2) Privacy-preserving Opening: In the privacy-preserving opening mode, the *client* expresses the **zkOpen** algorithm in a zkSNARK circuit $\mathcal{C}$ and executes the **zk.Prove** algorithm on the circuit $\mathcal{C}$ to obtain a proof π. Once the proof π is shared with the *proxy*, the *proxy* is able to verify a correct data opening by calling the **zk.Verify** function on the proof π and public inputs. The **zkOpen** algorithm takes as input private request and response plaintext chunks **pt**, computes and matches encrypted counter blocks **ecb**, and authenticates plaintext chunks by matching **ecb** values against public commitments s_{ECB}. Notice that the authentication of plaintext data remains independent of non-algebraic encryption function calls and depends on cheap xor computations instead. Efficiency is optimized as the commitment hash function H can make use of zkSNARK-friendly algebraic structures (e.g. H=MIMC) [22]. Last, the privacy-preserving opening mode verifies authenticated plaintext data against the statement ϕ by calling the function f_ϕ. This way, nothing except the statement validity is revealed to the *proxy*.

tpOpen(s_{ECB}, $\mathbf{I}^{\text{ECB}}$, i_q, **q**, **r**, **pt**, **ECB**):

1. $\mathbf{q}^{\text{chunks}} = \mathbf{q}[\mathbf{I}^{\text{ECB}}[:i_q]]$; $\mathbf{r}^{\text{chunks}} = \mathbf{r}[\mathbf{I}^{\text{ECB}}[i_q:]]$
2. $\mathbf{pt}^{\text{req}} = \mathbf{pt}[\mathbf{I}^{\text{ECB}}[:i_q]]$; $\mathbf{pt}^{\text{resp}} = \mathbf{pt}[\mathbf{I}^{\text{ECB}}[i_q:]]$
3. $\mathbf{ecb}^{\text{req}} = \mathbf{q}^{\text{chunks}} \oplus \mathbf{pt}^{\text{req}}$; $\mathbf{ecb}^{\text{resp}} = \mathbf{r}^{\text{chunks}} \oplus \mathbf{pt}^{\text{resp}}$
4. return: $H(\mathbf{ECB}) \stackrel{?}{=} s_{\text{ECB}} \,\&\, (\mathbf{ecb}^{\text{req}}||\mathbf{ecb}^{\text{resp}}) \stackrel{?}{=} \mathbf{ECB}$

zkOpen(**pt**, **ECB; s_{ECB}**, **q**, **r**, $\mathbf{I}^{\text{ECB}}$, i_q, m):

1. $\mathbf{q}^{\text{chunks}} = \mathbf{q}[\mathbf{I}^{\text{ECB}}[:i_q]]$; $\mathbf{r}^{\text{chunks}} = \mathbf{r}[\mathbf{I}^{\text{ECB}}[i_q:]]$
2. $\mathbf{pt}^{\text{req}} = \mathbf{pt}[\mathbf{I}^{\text{ECB}}[:i_q]]$; $\mathbf{pt}^{\text{resp}} = \mathbf{pt}[\mathbf{I}^{\text{ECB}}[i_q:]]$
3. $\mathbf{ecb}^{\text{req}} = \mathbf{q}^{\text{chunks}} \oplus \mathbf{pt}^{\text{req}}$; $\mathbf{ecb}^{\text{resp}} = \mathbf{r}^{\text{chunks}} \oplus \mathbf{pt}^{\text{resp}}$
4. $\mathbf{p}_{\text{ECB}} = f_{\bigoplus}(\mathbf{ecb}^{\text{req}}||\mathbf{ecb}^{\text{resp}}, m)$
5. assert: $s_{\text{ECB}} \stackrel{?}{=} (\mathbf{p}_{\text{ECB}}||H(\mathbf{ECB})) \& (\mathbf{ecb}^{\text{req}}||\mathbf{ecb}^{\text{resp}}) \stackrel{?}{=} \mathbf{ECB}$
6. assert: $1 \stackrel{?}{=} f_\phi(\mathbf{pt})$

Fig. 5. Transparent (**tpOpen**) and privacy-preserving (**zkOpen**) commitment opening algorithms, where H denotes the commitment function. The semicolon **;** separates inputs of the **zkOpen** function into private (left side) and public (right side) input arguments. The function f_ϕ validates input data against a public statement ϕ.

VI. Performance Evaluation

The evaluation comprises a description of the software stack in Section VI-A and provides a complete description and representation of benchmarks in the Section VI-B. Based on the performance metrics, we position the *Janus* type of TLS-oracle with regard to usability and security against other types of TLS-oracles in Section VII.

A. Implementation

We implemented the 3PHS by modifying the Golang *crypto/tls* standard library[2] and configured the NIST P-256 elliptic curve for the elliptic curve Diffie–Hellman exchange (ECDHE). Our proof of concept implementation configures TLS 1.3 with the cipher suite *TLS_AES_128_GCM_SHA256* and we implement the ECTF and Multiplicative to Additive (MtA) conversion algorithms with the Golang implementation of the Paillier cryptosystem [25]. For a coherent Golang implementation of the TLS-oracle *Janus*, we chose the *mpc* library [26] to access secure 2PC based on garbled circuits, and we chose the *gnark* framework [27] for the implementation of our ZKP circuits. The branch of *gnark* was configured to the latest available version *v0.9.0-alpha* and the underlying crypto library *gnark-crypto* was set to the version *v0.11.2*. We adjusted the *mpc* library to output single wire labels if we execute 2PC circuits in the context of HVZK proofs and we wrote all 2PC circuits in the *mpc*-specific multi-party computation language (MPCL). For a broader range of ZKP benchmarks, we executed ZKP circuits written in *gnark* in all available backend systems *groth16*, *plonk*, and *plonkFRI*, where *plonkFRI* does not depend on a trusted setup of cryptographic parameters.

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

TABLE II

SECURE COMPUTATION BENCHMARKS SEPARATED INTO OFFLINE AND ONLINE EXECUTION AND COMMUNICATION VALUES. WE SEPARATE SEMI-HONEST, MALICIOUSLY SECURE, AND HVZK 2PC SYSTEMS WITH DASHED LINES. WE INDICATE MALICIOUSLY SECURE AND HVZK 2PC SYSTEMS WITH AN ASTERISK $*$ AND THE SYMBOL Σ RESPECTIVELY. IN ZKP SYSTEMS, WE ABBREVIATE THE COMPILED CONSTRAINT SYSTEM (CCS), THE PROVER KEY (PK), THE VERIFIER KEY (VK), THE PUBLIC WITNESS (PW), AND THE PROOF π, WHERE PK AND VK TOGETHER FORM THE COMMON REFERENCE STRING (CRS) OF THE ZKP SYSTEM.

2PC Circuit	**Constraints** ($\times 10^6$)	**Execution Offline**	**Execution Online**	**Communication Offline**	**Communication Online**
$\mathcal{C}_{\text{XHTS}}$	3.14	215.56 ms	144 ms	34 MB	110 kB
$\mathcal{C}_{\text{k}^{m_1},\text{iv}}$	5.17	361.98 ms	242.41 ms	54.04 MB	178 kB
$\mathcal{C}^{256\text{ B}}_{\text{ECB}_{2+}}$ / $\mathcal{C}^{16\text{ kB}}_{\text{ECB}_{2+}}$	0.58 / 36.76	33.89 ms / 4.25 s	33.8 ms / 1.52 s	5.06 / 351.01 MB	58 kB / 2.02 MB
$*$ ECTF	-	-	212.96 ms	-	1.861 kB
$*$ $\mathcal{C}^{256\text{ B}}_{tag}$ / $\mathcal{C}^{2\text{ kB}}_{tag}$	4.04 / 29.01	285.98 ms / 2.42 s	492.24 ms / 3.78 s	52.06 / 378.02 MB	512 kB / 2 MB
Σ $\mathcal{C}^{256\text{ B req, 16 kB resp}}_{\text{HVZK}_{\text{private}}}$	82.1	11.4 s	3 s	794.05 MB	274.13 kB
Σ $\mathcal{C}^{256\text{ B req, 16 kB resp}}_{\text{HVZK}_{\text{transparent}}}$	79.12	12.68 s	5.19 s	754.04 MB	274.13 kB

		Execution Offline		**Execution Online**		**Communication Offline**			**Communication Online**	
ZKP Circuit	**Constraints** ($\times 10^3$)	Compile (s)	Setup (s)	Prove (s)	Verify (ms)	CCS, SRS (MB)	PK (MB)	VK (kB)	π (B)	PW (kB)
$\mathcal{C}^{256\text{ B}}_{\text{zkOpen}}$ / $\mathcal{C}^{16\text{ kB}}_{\text{zkOpen}}$	13.6 / 853.5	0.05 / 2.78	0.71 / 37.17	0.047 / 2.11	1.07 / 2.5	2.9 / 239.7	2.27 / 143.48	9.1 / 513	128	1.09 / 64.06

B. Performance

All performance benchmarks have been collected on a MacBook Pro configured with the Apple M1 Pro chip and 32 GB of memory. 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 and zkSNARK circuits, where execution times and communication overhead is further divided into offline and online benchmarks. Semi-honest 2PC circuits derive session secrets vis the circuits $\mathcal{C}_{\text{XHTS}}$, $\mathcal{C}_{\text{k}^{m_1},\text{iv}}$ in milliseconds and compute ECBs via the circuit $\mathcal{C}^{\text{X}}_{\text{ECB}_{2+}}$ for a 16 kB record in 1.52 seconds. For maliciously secure 2PC computations, the interesting fact to notice is that the AEAD tag circuit $\mathcal{C}_{tag}$ is efficient for small request sizes and scales sufficiently but not ideally for larger request sizes. The overhead in the circuit $\mathcal{C}_{tag}$ is introduced by the algebraic structure of the Galois field polynomials in GF(2^{128}), which is in conflict with the binary computation logic of boolean GCs. The related works [3], [8] propose a scalable and Oblivious Transfer (OT)-based computation of the AEAD tag, which we consider as future work to improve our implementation. Concerning HVZK circuits, we can see that the *transparent* protocol mode with circuit $\mathcal{C}_{\text{HVZK}_{\text{transparent}}}$ introduces more overhead compared to the *privacy-preserving* mode with circuit $\mathcal{C}_{\text{HVZK}_{\text{private}}}$. This behavior is expected because, the computation of the parity checksum on ECBs through $f_{\oplus}$ is less complex compared to computing a SHA256 hash on ECBs. Further, we generated the pseudorandom masks of the parity checksum inside the circuit $\mathcal{C}_{\text{HVZK}_{\text{private}}}$ based on a separate counter mode evaluation of AES128. This way, we received about double as many constraints and execution delays but saved the OT scheme to communicate 16.256 kB of data. With interest to run the *Janus* protocol as fast as possible, we evaluate the ZKP circuits of Table II using the *groth16* backend. As such, the privacy-preserving opening of TLS 1.3 data takes around 2.11 seconds for an entire record with 16 kB. In comparison, the transparent opening of 16 kB takes 0.05 milliseconds if the opening is evaluated with the SHA256 hash.

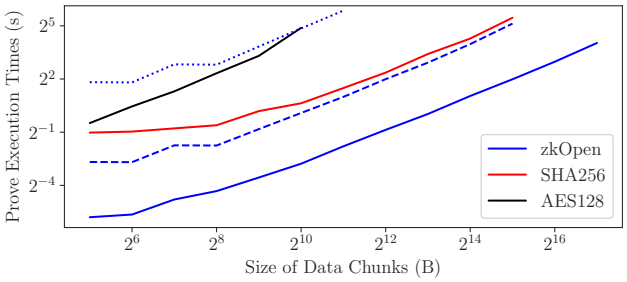

Fig. 6. Scalability analysis of zkSNARK circuits, where solid, dashed, and dotted lines indicate zero-knowledge proof computations based on the *groth16*, *plonk*, and *plonkFRI* backends respectively. Lines closer to the lower right corner are better and prove more data in less time.

The *groth16* backend achieves the best execution times among *gnark*-supported proof systems (cf. Section VI-B2) at the cost of requiring a trusted setup. Among zkSNARK proof systems for TLS-oracles, we expect *groth16* to remain relevant because of its maturity and existing efforts towards automating trusted setup ceremonies [28]. To clarify the behavior of different proof systems concerning different record sizes, we present a scalability analysis of the **zkOpen** circuit next.

2) Scalability Trade-off: We evaluate the **zkOpen** circuit by scaling circuit input sizes among different proof systems (cf. Figure 6). The scalability benchmarks show a linear relation between prove times and input sizes for every proof system. Within 4 seconds, the *groth16* backend proves 8 times more data than the *plonk* backend, and 320 times more data than the *plonkFRI* backend. To contrast the scalability benefits of the *Janus* protocol against the TLS 1.3 ECB commitment approach of the works [3], [8], [9] (e.g. decrypt-from-pad hash in [9]), and due to the fact that these related works have not provided a zkSNARK opening analysis yet, we compare the

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

	Communication (kb)				**Execution LAN (s)**			
Protocol	Offline	Handshake	Record	Post-record	Offline	Handshake	Record	Post-record
TLS 1.3	-	1.94	16.28	-	-	0.016	0.001	-
$Janus_{\text{private}}$	1.71 (GB)	223.8	764	338.3	53.21	0.634	0.769	5.15
$Janus_{\text{transparent}}$	1.28 (GB)	"	"	274.13	13.78	"	"	5.19

evaluation of the **zkOpen** circuit against an ECB commitment based on the SHA256 hash function. We deem the SHA256 hash as suitable because it relies on non-algebraic structures and can be efficiently computed in the 2PC approaches that are found in the related works. Thus, compared to an ECB commitment opening based on the SHA256 hash in the *groth16* proof system (cf. solid red line in Figure 6), the *Janus* protocol achieves 10 times faster opening benchmarks.

To compare our protocol against private and naive data opening circuits described in the early works [2], [4], we compare our benchmarks against our open sourced *gnark* implementation of AES128 (cf. solid black line in Figure 6). Data opening circuits which compute TLS encryption functions are compatible with TLS 1.2 and TLS 1.3 and, according to our analysis, achieve similar benchmarks as our protocol with with the *plonkFRI* zkSNARK backend. We further interpret our scalability results in the Section VII-B , where we position the *Janus* protocol in the context of TLS-oracles based on the achieved benchmarks.

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 request and a 16 kilo byte 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 ECBs after obtaining the session secrets of the *proxy*. Notice that in the handshake phase, the circuit $\mathcal{C}_{\text{XATS}}$ must be called twice to compute SHTS and CHTS. Concerning the post-record phase, the transparent protocol mode does not rely on any zkSNARK computations but instead, depends on the additional ECB hash computation which is verified in the HVZK circuit. 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 SHTS and certificate verification of the handshake phase takes 9.26 milliseconds and the witness extraction of the privacy-preserving *Janus* mode takes 4.47 milliseconds.

The resulting end-to-end benchmarks show that the ZKP preprocessing dominates offline execution times. The computation overhead of the *Janus* protocol mainly affects the post-record phase such that *clients* are able to efficiently collect data throughout the record phase. With regard to the size of privately opened data, our post-record execution timings establish new standards for privacy-preserving TLS-oracles. Our results serve as a comparison baseline for related works which, until now, miss out on a complete evaluation of privacy-preserving opening functions.

VII. DISCUSSION

The discussion presents related works, positions the *Janus* in the context of TLS-oracles with regard to supported properties, and summarizes remaining limitations.

A. Related Works

We outline our contributions in contrast to related works as follows. Our first protocol optimization follows the garble-then-prove paradigm of the work [20] but allows the execution of the garble-then-prove paradigm under different assumptions. In contrast to the work [20], TLS 1.3 allows the mutual verification and authentication of the SHTS secret. With access to the authenticated and correct SHTS secret, the *Janus* protocol verifies 2PC input correctness by matching 2PC inputs against the known parameter of SHTS. With this optimization, the *Janus* protocol does not rely on employing a semi-honest 2PC system with authenticated garbling, which is used in the work [20]. Instead, the *Janus* protocol can take advantage of the more efficient and highly optimized semi-honest 2PC systems based on boolean garbled circuits [13], which increases our overall protocol efficiency. We further notice that the reduction to a semi-honest 2PC setting without the requirement to employ an authenticated garbling technique allows the deployment of the HVZK proof system based on semi-honest 2PC [10]. The HVZK proof system of the work [10] can be used to efficiently evaluate non-algebraic algorithms. In our work, we exploit the benefits of the HVZK proof system to optimize the zkSNARK circuit complexity of the privacy-preserving data opening function. Related works, without access to the HVZK proof system propose privacy-preserving data openings based on zero-knowledge proof as follows. The work [9] leverages the structure of AEAD stream ciphers and demands clients to commit to stream cipher ECBs via a *pad commitment*. The works [3], [8] require parties to embed a commitment to encoded 2PC outputs into the 2PC interaction that computes ECBs. The last relevant approach converts authenticated session secrets into zkSNARK-friendly commitments to achieve efficient private data openings [20]. All approaches introduce interesting trade-offs, which our work partly improves upon.

Concerning the commitment to AEAD stream ciphers ECBs, the ZKP computation involves the computation of TLS

legacy algorithms (e.g., AES128), which are of non-algebraic structure. The related work [9] notices that legacy algorithms contribute to over 40% of client ZKP computation times and improves the efficiency of their protocol by putting the proof computation of the *pad commitment* into a pre-computing phase. Our work follows the approach to compute on stream cipher ECBs and introduces a commitment string construction based on a parity checksum. In detail, our parity checksum aggregates selected ECBs and interleaves the ECB commitment string with the disclosure of the parity checksum secret mask. As such, we verify the masked parity checksum on ECBs in the HVZK proof system and let clients prove the zkSNARK-friendly commitment string and the parity checksum in a zkSNARK proof system (cf. red and yellow boxes in Figure 1). Our decoupled verification of the commitment string with the help of the parity checksum yields an performance benefit by a factor $75x$ when opening 1 kB of data. Notice that our numbers are based on measuring non-optimized ZKP circuits of legacy algorithms.

Concerning the commitment to 2PC output encodings of the ECB computation, as suggested by the works [3], [8], the client faces the following limitation. The output encodings of ECBs prevent the client from accessing TLS plaintext 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 ECBs 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 ECBs 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 structure increases the complexity of privacy-preserving data openings, which scales with the amount of required commitments. Our work, in contrast, allows clients to access plaintext data before determining the selection of ECBs for the commitment string while remaining efficient.

The conversion of authenticated session secrets into zkSNARK-friendly commitments relies on an additive homomorphic commitment scheme [20]. In contrast, our protocol computes the commitment string based on zkSNARK-friendly hash-based commitment structures. According to state-of-the-art studies [24], hash-based commitment structures outperform additive homomorphic commitment structures which puts our work ahead.

B. Positioning of the Janus Protocol.

The most recent work on efficient TLS-oracles [20] achieves TLS 1.2 and TLS 1.3 compatibility and converts authenticated garbling parameters into a zkSNARK efficient Pedersen commitment. The Pedersen commitment can be used to open authenticated ciphertext chunks in a privacy-preserving data opening. Even though the work [20] does not benchmark a zero-knowledge opening of the Pedersen commitment, we expect their approach to achieve slightly less efficient privacy-preserving data openings. Because, our work does not depend on additive homomorphic commitment structures, which until today, remain less efficient compared to hash-based commitment structures [24]. Thus, in the context of TLS 1.3, we see the *Janus* protocol ahead in terms of scalable ZKP data openings compared to the work [20]. A detailed comparison of both protocols is a topic of future work.

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 talking 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.

C. Limitations

The *Janus* protocol is tailored to the characteristics and 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. The negative impact translates into a noticeable performance decrease because the *Janus* protocol depends on a maliciously secure 2PC interaction to compute request authentication tags. Last, due to the fact that the related works [3], [8], [20] withhold benchmarks of privacy-preserving TLS data openings, our performance results remain somewhat isolated and only allow us to make educated guesses about how our work behaves in comparison to other approaches (cf. Section VII-B).

VIII. Conclusion

In this work, we investigate how security characteristics and conditions of TLS 1.3 can be leveraged to complement security requirements of TLS-oracles. We find that the adversarial behavior in the two-party computation setting can be partly reduced to the assumption of semi-honest adversaries while maintaining a threat model with security against a malicious prover and verifier. Further, we decouple the structure of TLS 1.3 stream ciphers with a commitment scheme and a parity checksum such that TLS 1.3 integrity of the client can be verified efficiently in dedicated proof systems. Thus, our system sets new standards for private TLS 1.3 data proofs.

IX. Acknowledgements

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] D. Augot, S. Bordage, Y. El Housni, G. Fedak, and A. Simonet, "Zero-knowledge: trust and privacy on an industrial scale," 2022.

[24] L. Grassi, D. Khovratovich, R. Lüftenegger, C. Rechberger, M. Schofnegger, and R. Walch, "Reinforced concrete: A fast hash function for verifiable computation," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1323–1335.

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

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

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

[28] V. Nikolaenko, S. Ragsdale, J. Bonneau, and D. Boneh, "Powers-of-tau to the people: Decentralizing setup ceremonies," *Cryptology ePrint Archive*, 2022.

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

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

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

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

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

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

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

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

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

APPENDIX

A. 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 7), each party picks a secret randomness (s, v, p) and computes its encrypted representation (S, V, P). By sharing $V + P = X$ with the server in the CH, the server derives the session secret $Z_s = s \cdot X$, which corresponds to the TLS 1.3 secret DHE. When the server shares S in the SH, both the proxy and client derive their shared session secrets Z_v and Z_p respectively such that $Z_s = Z_v + Z_p$ holds. In the end, neither the client nor the verifier have full access to the DHE secret of the TLS handshake phase. The 3PHS works for both TLS versions but in Figure 7, we show a TLS 1.3-specific configuration based on the ECDHE, where the parameters (e.g. Z_p) are EC points structured as $P = (x, y)$.

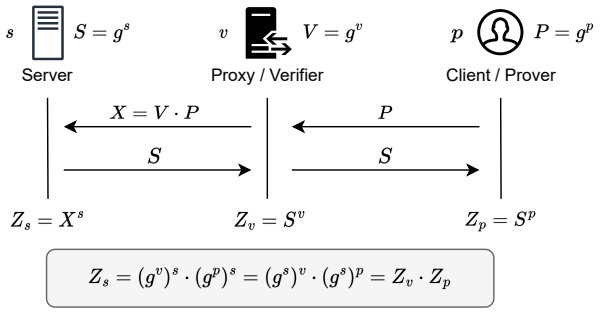

Fig. 7. Illustration of the 3PHS and exchanged cryptographic parameters between the server, the proxy, and the client. The gray box at the bottom indicates the relationship between shared client-side secrets Z_v and Z_p, which corresponds to the session secret Z_s of the server.

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

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

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

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

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

Both functions **hkdf.ext** and **hkdf.exp** internally use the **hmac** algorithm (cf. Formula 2), which takes in a key k, a bit string m, and generates a string which is indistinguishable from uniform random strings. The **hmac** algorithm requires a hash function H with input size b (e.g. b=64 if H=SHA256).

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

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

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

5) Secure Two-party Computation: Secure 2PC allows two mutually distrusting parties with private inputs x_1, x_2 to jointly compute a public function $f(x_1, x_2)$ without learning the private input of the counterparty. With that, secure 2PC counts as a special case of multi-party computation (MPC), with $m = 2$ parties and the adversary corrupting $t = 1$ parties [29]. The adversarial behavior model in 2PC protocols divides adversaries into semi-honest and malicious adversaries. Semi-honest adversaries honestly follow the protocol specification, whereas malicious adversaries arbitrarily deviate. In the following, we introduce secure 2PC protocols which are used in this work, and briefly introduce cryptographic constructions which are used to instantiate the secure 2PC protocols.

a) MtA Conversion based on Homomorphic Encryption: The secure 2PC MtA protocol converts multiplicative shares x, y into additive shares α, β such that $\alpha + \beta = x \cdot y = r$ yield the same result r. The MtA protocol exists in a vector form, which maps two vectors $\mathbf{x}, \mathbf{y}$, with a product $r = \mathbf{x} \cdot \mathbf{y}$, to two scalar values α, β, where the sum $r = \alpha + \beta$ is equal to the product r. The functionality of the vector MtA scheme can be instantiated based on Paillier additive Homomorphic Encryption (HE) [30]. 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 [31], with the following tuple of algorithms, where

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

The tuple of algorithms is supposed to be executed in the order where party p_1 first calls **mta.Setup** and **mta.Enc**. The function $E_k(z)$ is a Paillier encryption of message z under key k. After p_1 shares the public key pk_P and the vector of ciphertexts **c1** with party p_2, then p_2 calls **mta.Eval** and

ECTF between two parties p_1 and p_2.

inputs: $P_1 = (x_1, y_1)$ by p_1, $P_2 = (x_2, y_2)$ by p_2.
outputs: s_1 to p_1, s_2 to p_2.

p_1: (sk,pk)=**mta.Setup**(1^λ); send pk to p_2
p_1: $\rho_1 \xleftarrow{\$} \mathbb{Z}_p$; **c1**=**mta.Enc**$([-x_1,\rho_1], sk)$; send **c1** to p_2
p_2: $\rho_2 \xleftarrow{\$} \mathbb{Z}_p$;(c2,$\beta$)=**mta.Eval**(**c1**,$[\rho_2,x_2]$,$pk$);$\delta_2=x_2\cdot\rho_2+\beta$; send (c2,$\delta_2$) to p_1
p_1: α=**mta.Dec**(c2,sk);$\delta_1=-x_1\cdot\rho_1+\alpha$;$\delta=\delta_1+\delta_2$;$\eta_1=\rho_1\cdot\delta^{-1}$; **c1**=**mta.Enc**$([-y_1,\eta_1],sk)$; send (**c1**,$\delta_1$) to p_2
p_2: $\delta=\delta_1+\delta_2$; $\eta_2=\rho_2\cdot\delta^{-1}$; (c2, β)=**mta.Eval**(**c1**,$[\eta_2,y_2]$,pk); $\lambda_2=y_2\cdot\eta_2+\beta$; send c2 to p_1
p_1: α=**mta.Dec**(c2,sk); $\lambda_1=-y_1\cdot\eta_1+\alpha$; **c1**=**mta.Enc**$([\lambda_1],sk)$; send **c1** to p_2
p_2: (c2,β)=**mta.Eval**(**c1**, $[\lambda_2]$, pk); $s_2 = 2\cdot\beta + \lambda_2^2 - x_2$; send c2 to p_1
p_1: α=**mta.Dec**(c2,sk); $s_1 = 2\cdot\alpha + \lambda_1^2 - x_1$

Fig. 8. The ECTF algorithm converts multiplicative shares in form of EC point x-coordinates from points $P_1, P_2 \in EC(\mathbb{F}_p)$ to additive shares $s_1, s_2 \in \mathbb{F}_p$. It holds that $s_1 + s_2 = x$, where x is the coordinate of the EC point $P_1 + P_2$.

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

b) ECTF Conversion: The ECTF algorithm is a secure 2PC protocol and converts multiplicative shares of two EC x-coordinates into additive shares [2], [4]. Figure 8 shows the computation sequence of the ECTF protocol which makes use the vector MtA algorithm defined in Section A5a. By running the ECTF protocol, two parties p_1 and p_2, with EC points P1, P2 as respective private inputs, mutually obtain additive shares s_1 and s_2, which sum to the x-coordinate of the EC points sum P1+P2. TLS-oracles use the ECTF protocol to transform the client-side EC secret shares Z_v and Z_p into additive shares s_v and s_p [2], [4]. Since the relation $s_v + s_p = x$ for $(x, y) = Z_s$ holds, it becomes possible to follow the TLS specification by using secure 2PC based on boolean garbled circuits with bitwise additive shares as input.

c) Oblivious Transfer: Secure 2PC based on boolean GCs depends on the 1-out-of-2 OT_2^1 sub protocol to secretly exchange input parameters of the circuit [32]. The OT_2^1 involves two parties where party p_1 sends two messages m_1, m_2 to party p_2 and does not learn which of the two messages m_b is revealed to party p_2. Party p_2 inputs a secret bit b which decides the selection of the message m_b. An OT scheme is defined by a tuple of algorithms, where

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

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

d) Semi-honest 2PC *with Garbled Circuits:* Our description of a GC scheme requires additional notations. We express a GC scheme with the parameters $\mathcal{GC} = \{d, l, w, T_{d\text{-}l}, T_{l\text{-}w}, T_{l\text{-}w}^G, \mathcal{C}_G\}$, where d is a bit string of data, which we further divide into input, intermediate, and output strings $d = \{d_{\text{in}}, d_{\text{interm}}, d_{\text{out}}\}$. Since input data is secret shared between two parties, we divide input bit strings into $d_{\text{in}} = d_{\text{in}}^{p_1} || d_{\text{in}}^{p_2}$. We divide labels into input, intermediate, and output labels $l = \{l_{\text{in}}, l_{\text{interm}}, l_{\text{out}}\}$ and break down input labels into $l_{\text{in}} = \{l_{\text{in}}^{d_{\text{in}}^{p_1}}, l_{\text{in}}^{\neg d_{\text{in}}^{p_1}}, l_{\text{in}}^{d_{\text{in}}^{p_2}}, l_{\text{in}}^{\neg d_{\text{in}}^{p_2}}\}$, where labels correspond to previously introduced bit strings. Last, we divide bit-to-label tables into encoding, evaluation, and decoding mappings $T_{d\text{-}l} = \{T^{\text{enc}}_{d_{\text{in}}\text{-}l_{\text{in}}}, T^{\text{eval}}_{d_{\text{interm}}\text{-}l_{\text{interm}}}, T^{\text{dec}}_{l_{\text{out}}\text{-}d_{\text{out}}}\}$. We denote the boolean garbled circuits as $\mathcal{C}_G$, with boolean logic gates AND and XOR, and wires w. The tables $T_{l\text{-}w}, T_{l\text{-}w}^G$ refer to label-to-wire and garbled label-to-wire mappings respectively. The garbled table $T_{l\text{-}w}^G$ encrypts every output wire label under input wire label combinations of the same gate. Finally, we define secure 2PC based on boolean garbled circuits by extending our OT definition of Section A5c with the tuple of algorithms, where

- **gc.Setup**$(1^\lambda) \rightarrow (\text{pp}_{\text{GC}})$ takes in the security parameter λ and outputs public parameters pp_{GC}.
- **gc.Garble**$(\text{pp}_{\text{GC}}, \mathcal{C}_G, d_{\text{in}}) \rightarrow (l, T_{\text{l-w}}, T_{\text{l-w}}^G, T_{d\text{-}l})$ takes as input pp_{GC}, a boolean circuit $\mathcal{C}_G$, the input bit string d_{in}, and randomly samples $r_G, l \xleftarrow{\$} \mathbb{Z}_n$. The output consists of labels l, the undistorted and garbled label-to-wire tables $T_{\text{l-w}}, T_{\text{l-w}}^G$, and the bit-to-label table $T_{d\text{-}l}$.
- **gc.Evaluate**$(\text{pp}_{\text{GC}}, l_{\text{in}}^{d_{\text{in}}^{p_1}}, l_{\text{in}}^{d_{\text{in}}^{p_2}}, T_{\text{l-w}}^G, T^{\text{dec}}_{l_{\text{out}}\text{-}d_{\text{out}}}) \rightarrow (d_{\text{out}})$ takes in pp_{GC}, input labels $l_{\text{in}}^{d_{\text{in}}^{p_1}}, l_{\text{in}}^{d_{\text{in}}^{p_2}}$, the garbled label-to-wire table $T_{l\text{-}w}^G$, a label-to-bit decoding $T^{\text{dec}}_{l_{\text{out}}\text{-}d_{\text{out}}}$, and outputs the bit string d_{out}.

On a high-level, a 2PC system based on boolean garbled circuits involve a party p_1 as the garbler and party p_2 as the evaluator. Party p_1 calls **gc.Setup** and **gc.Garble**, where p_1 loops over every wire of the circuit $\mathcal{C}_G$ and assigns two

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

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

e) Malicious Secure 2PC *with Dual Execution:* The malicious secure 2PC paradigm of the work [3], [14] runs two instances of a semi-honest 2PC evaluation of a circuit $\mathcal{C}$, where the *proxy* and *client* successively act as the garbler. Afterwards, the protocol runs a secure validation phase and with that a mutual output label equality check. The idea behind the secure validation phase is as follows. If the *client* as the evaluator obtains output labels l_c and bits b from a correctly garbled circuit of the *proxy*, then the *client* knows which output labels l_p according to b a *proxy* must evaluate on a correctly garbled circuit of the *client*. Thus, if the *client* shares a commitment in form of a hash $H(l_p||l_c)$ with the *proxy* after the first circuit evaluation, and the proxy returns the same hash $H(l_p||l_c)$ after the second circuit evaluation, then the *client* is convinced of a correct circuit garbling. Because, if the *proxy* incorrectly garbles a circuit, then the *client* obtains wrong bits b'. And if the *client* correctly garbles a circuit, the *proxy* obtains correct bits b. The incorrect bits b' lead the *client* to a selection of labels l'_c and l'_p and the correct bits b lead the *proxy* to a correct selection of $l_p \neq l'_p$. However, since the *proxy* does not know which incorrect output labels l'_c the *client* evaluated, it cannot predict either of the incorrect output labels l'_c, l'_p, which are expected by the *client*. Only when using a correctly garbled circuit, the same bits b lead both parties to the same labels which have been separately samples at random.

6) 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.

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

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

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

7) Zero-knowledge Proof Systems: Proof systems allow a prover p to convince a verifier v of whether or not a statement is true. In theory, proof systems rely on a NP language $\mathcal{L}$ and the existence of an algorithm $R_{\mathcal{L}}$, which can decide in polynomial time if w is a valid proof for the statement $x \in \mathcal{L}$ by evaluating $R_{\mathcal{L}}(x, w) \stackrel{?}{=} 1$. The assumption is that for any statement $x \in \mathcal{L}$, there exist a valid witness w and no witness exists for statements $x \notin \mathcal{L}$. In order for proof systems to work, three properties of *completeness*, *soundness*, and *zero-knowledge* must hold. Completeness ensures that an honest prover convinces an honest verifier by presenting a valid witness for a statement. Soundness guarantees that a cheating prover cannot convince a honest verifier by presenting an invalid witness for a statement. Zero-knowledge guarantees that a malicious verifier does not learn anything except the validity of the statement. Beyond the three properties, the special notion of HVZK considers a semi-honest verifier, and, when applied, improves efficiency of the proof system [33], [34].

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

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

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

a) Zero-Knowledge Succinct Non-Interactive Argument of Knowledge: A zkSNARK proof system is a zero-knowledge proof system, where the four properties of *succinctness*, *non-interactivity*, computational sound *arguments*, and witness *knowledge* hold [35]. 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 [36]), then the prover is able to

convince the verifier by sending a single message. Computational sound arguments guarantee soundness in the zkSNARK system if provers are computationally bounded. Last, the knowledge property ensures that provers must know a witness in order to construct a proof.

b) HVZK based on Garbled Circuits: Interestingly, zero-knowledge is a subset of secure 2PC. For instance, a ZKP can be computed by using GC-based 2PC, where only one party inputs private data. In this work, we make use of the HVZK proofs built with boolean GCs in a semi-honest 2PC setting [10]. 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 either yields zero or one. By running the OT protocol for each witness bit, the evaluator as the prover obtains garbled input labels without revealing any witness bits to the garbler. After the prover evaluates the GC and returns the wire label corresponding to a one, 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 [10].

B. Security Analysis

The security analysis audits the properties of *session-integrity*, *session-confidentiality*, *session-authenticity*, and *data-provenance* under the assumption that MITM attacks are feasible. Other system goals come for free by following established approaches of TLS-oracles. For our theorems, we assume that the security guarantees provided by TLS 1.3 [11] hold.

1) Handshake Phase: The *Janus* protocol ensures that, throughout the TLS handshake phase, the properties of *session-confidentiality*, *session-integrity*, and *session-authenticity* hold.

a) Session Confidentiality: In the handshake phase, semi-honest 2PC circuits are executed as follows. Before evaluating the circuit $\mathcal{C}_{\text{XHTS}}$ with labels that output the secret SHTS, the *client* expects the encrypted SF message and discloses the 2PC output SHTS if the verification of SF and the *server* certificate succeeds. The evaluation of $\mathcal{C}_{\text{XHTS}}$ for CHTS happens after the exchange of SHTS.

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

Proof 1. If the adversary maliciously garbles $\mathcal{C}_{\text{SHTS}}$ such that the output yields SHTS=0, then the adversary is able to generate server-side handshake traffic keys and present a forged but conform encryption of the SF message to the *client*. The adversary must further forge the server certificate (SC) message and the SCV message to force the *client* into passing the SHTS validation. However, the adversary cannot learn any information on the *client* secret share s_2 because receiving SHTS=0 does not leak any information of s_2. If the adversary maliciously garbles $\mathcal{C}_{\text{SHTS}}$ and sets the output SHTS to s_2 or to an information leakage L_{s_2} on s_2 (e.g. statistic), then the adversary must correctly guess L_{s_2}, s_2 or find a hash collision on the inputs L_{s_2}, s_2. Otherwise, the adversary cannot generate forged messages which pass the SHTS validation at the *client*. The key independence property of TLS 1.3 prevents the adversary from evaluating a guess on s_2 with the help of L_{s_2}, which reduces the challenge for the adversary to the hardness of guessing s_2. Thus, with a deployment of a collision resistant hash function and a OT scheme which mutually protects the 2PC inputs, the property of *session-confidentiality* is preserved when evaluating $\mathcal{C}_{\text{SHTS}}$ according to the *Janus* protocol.

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

b) Session Integrity: Handshake *session-integrity* prevents an adversary from deviating the TLS 1.3 handshake specification and provides honestly operating parties with an identifiable abort option when detecting misbehavior of the adversary.

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

Proof 2. Injecting false inputs into the semi-honest 2PC circuit $\mathcal{C}_{\text{XHTS}}$ leads to a computation of SHTS' and CHTS', which deviate from server-side secrets SHTS and CHTS. As a consequence, the *server* is incapable of processing non-compliant AEAD parameters which have been derived under false session secrets. The same session abort happens if the adversary modifies the traffic transcript without full access to the secret shared client session.

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

Theorem 3. *Under the honest server assumption and a secure digital signature scheme, the Janus protocol achieves session-authenticity against a probabilistic polynomial time adversary $\mathcal{A}$ which mounts MITM attacks and replays previously established TLS 1.3 sessions.*

Proof 3. 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* immediately shares server-side handshake messages back to the *client*. If the *proxy* intercepts correct SF, SCV, and SC messages before participating in the 2PC computation of SHTS, any subsequent forged but compliant server-side handshake messages of the adversary are not accepted by the *proxy* anymore. Hence, the adversary must disclose the correct SHTS secret to proceed in our protocol. The *proxy* further detects forged server-side messages because the adversary cannot share compliant server-side handshake messages before evaluating SHTS. Our system model prevents the adversary from forging PKI certificates without access to the server IP address. Hence, the secure digital signature scheme prevents the adversary from compromising the *server* key pair which has been authenticated by the PKI certificate. TLS 1.3 replay attacks are infeasible because every TLS 1.3 *Janus* session injects new randomness via the updated CH message key share at the *proxy*. Thus, *session-authenticity* is preserved if the *proxy* 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.

2) Record Phase: The security analysis of the record phase investigates if the security properties of *session-confidentiality*, *session-integrity*, and *data-provenance* hold throughout the TLS 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* or yield information on AEAD parameters which leak private record data of the *client* (e.g. ECBs 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 4. *Under a maliciously secure OT scheme, a secure commitment scheme, and a HVZK proof system leaking a single bit, the Janus protocol maintains session-confidentiality during the TLS 1.3 record phase against a probabilistic polynomial time adversary $\mathcal{A}$, which maliciously garbles semi-honest* 2PC *circuits or acts as a malicious verifier.*

Proof 4. The adversary $\mathcal{A}_1$, acting as a *client*, is not able to learn any encrypted counter blocks of the type ECB_{tag} before the *proxy* intercepts a corresponding pair of a ciphertext and authentication tag. Because, the *proxy* discloses TLS 1.3 session secrets after the transcript traffic has been recorded. Further, the deployment of malicious secure 2PC system to compute request authentication tags does not leak information on secret inputs or AEAD parameters to $\mathcal{A}_1$. Thus, $\mathcal{A}_1$ is neither able to predict a forged authentication tag and, with that, gain the ability to inject arbitrary data into the protocol, nor can $\mathcal{A}_1$ learn *client* secret shares because the malicious secure OT scheme protects 2PC inputs of the *proxy*.

Vice versa, an adversary $\mathcal{A}_2$, acting as the *proxy*, cannot access record phase 2PC inputs due to the employed maliciously secure OT scheme. By receiving the parity checksum, and with knowledge of the random masks, $\mathcal{A}_2$ cannot access any ECB bit string because the xor aggregation only disclose xor results of ECBs to $\mathcal{A}_2$. If $\mathcal{A}_2$ maliciously garbles the HVZK circuit $\mathcal{C}^{\text{HVZK}}$, then a single bit can be leaked to $\mathcal{A}_2$. However, the circuit $\mathcal{C}^{\text{HVZK}}$ verifies session secrets shares, ECBs, or intermediate checksum parameters which all contain sufficient randomness such that leaking a bit is bearable [21]. On the contrary, zkSNARK proof systems, which are used to compute privacy-preserving data openings, do not leak any bits when opening private record data. Thus, *session-confidentiality* hold throughout the *Janus* protocol against $\mathcal{A}_2$ and until the secret disclosure of *proxy* secrets towards $\mathcal{A}_1$.

b) Session Integrity: To compromise *session-integrity* in the record phase, the adversary intends to inject arbitrary AEAD parameters into the HVZK circuit $\mathcal{C}^{\text{HVZK}}_{s_{\text{ECB}}}$ such that the verifier accepts an incorrect derivation of the parity checksum. Further, the adversary tries to share false 2PC outputs to the *proxy* and provides false inputs to 2PC circuits in order to extract secret session information of the counterparty.

Theorem 5. *Under a HVZK proof system, a maliciously secure OT scheme, a maliciously secure* 2PC *system, randomly samples bit string masks, and a secure commitment scheme, the Janus protocol preserves session-integrity against a probabilistic polynomial time adversary $\mathcal{A}$ which acts as a cheating prover, malicious garbler, or injects false TLS values into* 2PC *circuits.*

Proof 5. To prove *session-integrity* in the record phase, we introduce two adversaries $\mathcal{A}_{1,2}$. The adversary $\mathcal{A}_1$ takes the role of the *proxy* and maliciously garbles the 2PC circuits $\mathcal{C}_{(\text{k}^{m_1},\text{iv})}$, $\mathcal{C}_{\text{ECB}_{2^+}}$, and $\mathcal{C}_{tag}$ or provides false 2PC inputs. After recording all record phase transcript pairs ($\mathbf{x}$, t^x), $\mathcal{A}_1$ shares all TLS session secret shares with the *client* and shares the garbled truth table after receiving a commitment of the HVZK output label with the *client*. The *client* detects cheating behavior of $\mathcal{A}_1$ by locally recomputing and matching TLS 1.3 session parameters against all 2PC outputs. Validity of recomputed parameters is guaranteed because the *client* verifies recomputed values against the SHTS parameter which has been authenticated by the *server*. Thus, any incorrect 2PC input provisions of $\mathcal{A}_1$ lead to protocol aborts which guarantees *session-integrity* of $\mathcal{A}_1$. Neither $\mathcal{A}_1$ nor the adversary $\mathcal{A}_2$, who acts as the *client*, cannot provide false inputs or maliciously garble the circuit $\mathcal{C}_{tag}$, because $\mathcal{C}_{tag}$ is evaluated in the maliciously secure 2PC system. Malicious garbling is detected by the *client* because the garbled circuits are re-computed or truth tables allow for honest garbling verification at the *client*.

Concerning $\mathcal{A}_2$, any TLS 1.3 protocol deviation is caught by the requirement on $\mathcal{A}_2$ to share a HVZK proof of the circuit $\mathcal{C}_{\text{HVZK}}$ with the *proxy*. In the record phase, the authenticated

and correct SHTS parameter is publicly known and the *proxy* verifies input correctness of $\mathcal{A}_2$ in $\mathcal{C}_{\text{HVZK}}$ by matching a computation trace from input secret shares against SHTS. After the assertion of correct inputs, the circuit computes all encrypted counter blocks **ECB** and matches derived $\mathbf{ECB}_{tag}$ against public input $\mathbf{ECB}_{tag}$ vectors. The $\mathbf{ECB}_{tag}$ verification ensures that the circuit $\mathcal{C}_{tag}$ has been evaluated with valid inputs that are conform to the TLS session. Thus, $\mathcal{A}_2$ cannot input incorrect values into $\mathcal{C}_{tag}$. The verification of $\mathbf{ECB}_{tag}$ vectors further ensures that intercepted request and response records preserve integrity regarding the TLS session such that $\mathcal{A}_2$ cannot forge transcript data against the *proxy*.

Concerning the computation of the parity checksum, $\mathcal{A}_2$ is able to commit to arbitrary **ECB**' values and pass in valid **ECB** values into the circuit $\mathcal{C}_{\text{HVZK}}$ such that the circuit succeeds in verifying the parity checksum $p_{\mathbf{ECB}}$. Notice that *session-integrity* and *session-authenticity* hold on the parity checksum, if the circuit $\mathcal{C}_{\text{HVZK}}$ successfully verified the parity checksum. If *session-integrity* and *session-authenticity* of a parity checksum $p_{\mathbf{ECB}}$ hold, $\mathcal{A}_2$ cannot open arbitrarily committed **ECB**' values in the zkSNARK circuit because the parity checksum only verifies for a correct set of **ECB** values (cf. Section B2c). Thus, the parity checksum enforces *session-integrity* on $\mathcal{A}_2$ throughout the record phase.

c) Data Provenance: To compromise *data-provenance*, the adversary tries to convince a verifier of an invalid computation trace from presented data towards TLS 1.3 session parameters, which have been verified regarding authenticity and integrity.

Theorem 6. *Under a secure zkSNARK proof system, and a secure commitment scheme, and randomly samples bit string masks, data-provenance of the Janus protocol holds against a probabilistic polynomial time adversary $\mathcal{A}$, which, acting as a cheating prover, evaluates a plaintext **pt** against a pair of a ciphertext and commitment string (**ct**, s_{ECB}) with session-authenticity and session-integrity.*

Proof 6. The input of private encrypted counter blocks **ECB** is fully determined by the zkSNARK circuit because the circuit matches all individual bits inside **ECB** against a commitment string s_{ECB} with *session-authenticity* and *session-integrity*. Due to the *binding* property of the employed commitment string, the probability of the adversary to find a collision of ECB$'$ and ECB against the hash in s_{ECB} is negligible. Further, if the public bit string masks combined with private **ECB** values evaluate to the parity checksum in s_{ECB}, then the **ECB** values fulfil *session-integrity*. The reasoning is as follows. The adversary can guess an xor evaluation of a bit string mask m' with an arbitrary ECB' value that matches the xor evaluation of a valid bit string mask m and a valid ECB with negligible probability. Even though the adversary has access to the vector **ECB**, guessing a masked ECB without the mask has negligible success probability (e.g., 2^{-127} for 16 byte ECB chunks). Thus, since the zkSNARK circuit fully determines the verification trace of **ECB** values, and matches the computation of plaintext chunks **pt** xored with previously verified **ECB** values against authenticated ciphertext chunks **ct**, *data-provenance* of **pt** is guaranteed.

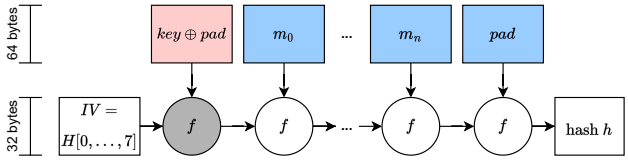

Fig. 9. Merkle-Damgård structure of the SHA256 hash function. Values marked in red indicate private input whereas blue background indicates public input. To protect a secret *key*, the prover must compute the first f (grey background) in circuit. All remaining intermediate hash values f (white background) can be computed out of circuit by the verifier and checked against the public input hash value h.

C. HVZK Circuit Optimization

The TLS 1.3 key derivation can be optimized by leveraging the security provided by the Merkle-Damgård structure. The Merkle-Damgård structure is used to compress input data to a fixed size output in hash algorithms (e.g. SHA256). If TLS 1.3 is configured with the cipher suite `TLS_AES_128_GCM_SHA256`, then the Merkle-Damgård repetitively appears during the TLS key derivation function `hkdf.extr` and `hkdf.exp` because the key derivation functions internally call **hmac**, which in turn, calls SHA256 and with that the Merkle-Damgård structure.

1) Merkle-Damgård Structure: In a scenario where **hmac** is called in TLS 1.3 and TLS 1.3 is configured with `TLS_AES_128_GCM_SHA256`, the concatenation of the inner hash $H((K' \oplus ipad)||m)$ (32 bytes) and $K' \oplus opad$ (64 bytes) yields a 96 byte output, which in turn, is the input to the outer hash function (cf. Formula 2). The input to the inner hash function is of size $64 + \text{len}(m)$ bytes. Thus, both hash input sizes in HMAC are above 64 bytes. If the hash input of SHA256 is above 64 bytes, SHA256 applies the Merkle-Damgård structure which repeats calls to an internal compression blockcipher f to reduce the input to a fixed sized output. The compressing blockcipher SHACAL-2 of SHA256 uses 64 computation rounds to hide its input and has not been broken [37]. Thus depending on whether the inner or outer hash is computed, the first call of the one-way compression blockcipher inside SHA256 already hides inputs $(K \oplus ipad)$ or $(K \oplus opad)$ of size 64 bytes and with that, hides the secret K of the prover [4]. As a result, the output of the compressing blockcipher in SHA256 can be used as public input to reduce the HVZK circuit complexity.

Figure 9 shows the case which applies in a ZKP circuit to compute the HMAC inner hash $H_{inner} = H((K' \oplus ipad)||m)$, where e.g. $m = \mathbf{H}_2$ is publicly known input. If m is publicly known by the verifier, the prover can compute the grey f and disclose it to the verifier, which computes the remaining part of the hash out of circuit. The same optimization of SHA256 is feasible when computing the outer hash $H_{\text{outer}} = H((K' \oplus opad)||\mathbf{H}_{\text{inner}})$. Thus, proving HMAC in a ZKP takes two evaluations of the SHACAL-2 compression function f if the message input m is publicly known.

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