ORIGO: Proving Provenance of Sensitive Data with Constant Communication

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

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

Yinnan Wu
yinnan.wu@tum.de
Technical University of Munich
Munich, Germany

Arthur Gervais
arthur@gervais.cc
University College London
London, United Kingdom

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

ABSTRACT

Transport Layer Security (TLS) is foundational for safeguarding client-server communication. However, it does not extend integrity guarantees to third-party verification of data authenticity. If a client wants to present data obtained from a server, it cannot convince any other party that the data has not been tampered with.

TLS oracles ensure data authenticity beyond the client-server TLS connection, such that clients can obtain data from a server and ensure provenance to any third party, without server-side modifications. Generally, a TLS oracle involves a third party, the verifier, in a TLS session to verify that the data obtained by the client is accurate. Existing protocols for TLS oracles are communication-heavy, as they rely on interactive protocols. We present ORIGO, a TLS oracle with constant communication. Similar to prior work, ORIGO introduces a third party in a TLS session, and provides a protocol to ensure the authenticity of data transmitted in a TLS session, without forfeiting its confidentiality. Compared to prior work, we rely on intricate details specific to TLS 1.3, which allow us to prove correct key derivation, authentication and encryption within a Zero Knowledge Proof (ZKP). This, combined with optimizations for TLS 1.3, leads to an efficient protocol with constant communication in the online phase. Our work reduces online communication by 375× and online runtime by up to 4.6×, compared to prior work.

1 INTRODUCTION

Ensuring sovereignty over digital content is imperative in contemporary computing systems and applications [31]. Recent endeavors focus on cryptographic verification of digital identities [49, 52], combating disinformation [42], and enforcing access control [43]. Nonetheless, these solutions are hampered by a bootstrapping dilemma; they are rendered ineffective without a centralized root of trust that ensures content authenticity.

Recent work introduces *TLS oracles*, which aim to solve this problem by cryptographically affirming the origin of digital content [67]. To do so, these solutions augment a standard TLS connection with a third party, such that the third party is oblivious to the content sent between a client and a server. The third party operates independently of the content-serving server and acts as a *verifier*, solely verifying the data exchanged. This approach empowers clients, allowing them to secure attestations for their digital content without demanding for clearance from a central application server.

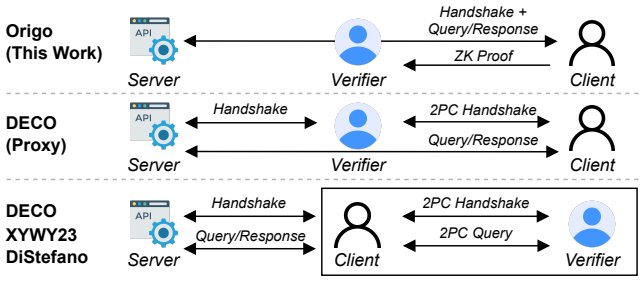

Figure 1: Systematization of system designs for TLS Oracles. Origo expands the existing set of systems by ensuring ciphertext integrity in TLS 1.3 with ZKPs.

One critical application of such a system are third-party logins. Numerous websites permit user authentication via third-party credentials, bypassing the need for account creation. However, these applications frequently overshare data and lack the capability for selective content disclosure. Additionally, users are forced to depend on the third-party application server for successful access delegation—a reliance often unmet due to these applications' disinclination to share user data. TLS oracles circumvent these issues, allowing users to acquire verified content and credentials without revealing their intentions to the application server. For example, the user can obtain an attestation from the third party in the TLS session, attesting that the information indeed originates from, e.g., a governmental server. Any website can now verify the attestation, the user does not need any further interaction with the government server to ensure that the presented information is authentic.

Previous approaches to TLS oracles that do not require server-side changes can be divided based on whether they operate with the verifier as an external entity [22, 23, 63, 67] or with a verifier as a proxy in between the client and the server [46, 67] (cf. Figure 1). All solutions aim to solve the same set of challenges — ensuring that the message presented by the client is consistent with the response sent by the server, ensuring that the verifier cannot provide an invalid response to the client, and providing confidentiality towards the verifier. To provide each of these guarantees, all protocols operate

*Both authors contributed equally to this research.

in roughly the same manner. First, a secure connection to the server is established by executing the TLS handshake and key derivation in 2PC. Second, the client and verifier collaboratively send the request to the server, whereas the response is provided to the verifier *before* the client obtains the full decryption key. Third, the client generates a ZKP to prove a statement on the private payload. In the proxy mode, post-handshake 2PC protocols can be discarded [67]. Notably, all previous approaches require an interactive protocol, with communication linear in the number of gates in the 2PC circuit, for non-repetitive computations [62].

In this work, we introduce ORIGO, a TLS oracle with constant communication, independent of the size of the data requested from an application server. We leverage that in TLS 1.3, the prevalent version of TLS used to date, the IV for Authenticated Encryption with Associated Data (AEAD) is derived from the traffic secret to facilitate a TLS oracle that does not demand for any communication intensive 2PC. By doing so, we can use a simple Succinct Non-interactive Argument of Knowledge (SNARK) to prove compliance with TLS 1.3, ensuring the same properties as achieved in previous work, under the assumption of a weaker network adversary that is unable to intercept traffic between the proxy verifier and server. With ORIGO, clients can obtain attestations on resource constrained devices, even in areas with low connectivity. Attestations can successively be used with arbitrary web applications and blockchains. Additionally, we provide an efficient transformation of AEAD ciphertexts to SNARK friendly commitments, enabling clients to prove disjunct statements on data obtained — without demanding re-execution of the protocol.

Technical Overview. Thus far, the proxy setting for TLS oracles has only been sparsely described in academic works (cf. Figure 1). Therefore, we set out to clarify security assumptions, and possible attacks to determine efficient solutions for data provenance.

Security in the Proxy Setting. We start by examining the security assumptions of TLS in the proxy model. Zhang *et. al* first introduce the proxy setting in the appendix of their seminal work [67]. They note that in the proxy setting, one has to assume a weaker network adversary, as the client could perform a machine-in-the-middle (MITM) attack between the client and the server. We find that this assumption is indeed a necessity, as none of the server-side messages are signed and TLS employs symmetric encryption. As a result, a MITM client can always simulate an interaction with an honest server. Further, Zhang *et. al* note that the handshake remains to be executed with *three* parties, as AES-GCM is non-committing as proven by Grubbs *et. al* [39]. Although AES-GCM is non-committing, the attack as described by Grubbs *et. al* is not trivially applicable, as a winning adversary needs to be able to choose an arbitrary ciphertext to win the security game. In Section 4, we introduce the notion of *strong receiver binding*, which applies to the proxy setting in TLS oracles. We further introduce an attack, showing that AES-GCM is *not* strong receiver binding in TLS 1.3.

SNARKs for TLS 1.3. Our protocol shows that a SNARK can be used to enforce strong receiver binding in the proxy setting of a TLS oracle in TLS 1.3. We derive a set of tailored optimizations that are specific to TLS 1.3 and minimize the number of constraints in the SNARK circuit. In comparison to the work of Grubbs *et. al* [38], our circuit requires $\sim 3\times$ less constraints to derive a key in TLS 1.3.

Client-side evidence generation. We detail an extension to our protocol, which allows for expanded use-cases besides the issuance of trivial credential. We ensure that clients can generate evidence of ciphertext integrity by efficiently transforming AEAD plaintexts to hiding and binding commitments in TLS oracles. To do so, we let the client expand block-level nonces from the initialization vector derived from the TLS handshake in TLS 1.3. We facilitate efficiency by instantiating the HKDF.expand function with a SNARK friendly MiMC hash, and instantiate the commitment to TLS records with a salted hash. The generation of commited values augments existing circuits as an extension. We instantiate the implementation of MiMC hashes with Groth16 as the outer SNARK and recurse the inner verifier of the GKR protocol. Previous work shows that this optimization leads an order of magnitude improvement in in-circuit hash computation [16]. We show that generating evidence for a 1kB TLS record only takes an additional 0.12 seconds when optimized with the recursive construction by Belling *et.al* [16].

In summary, our contributions are as follows:

- **A novel TLS oracle construction.** We introduce ORIGO, a protocol designed for proving data provenance with constant communication in TLS 1.3. We observe that in TLS 1.3, a simple SNARK is sufficient to ensure integrity when instantiating the third party in the TLS exchange as a proxy.
- **A novel attack on TLS 1.3.** We show that due to derivation of the traffic encryption key from the TLS handshake transcript, a malicious client could arbitrarily manipulate the plaintext, whilst retaining authenticity. We introduce the notion of strong receiver binding, and an attack that shows that the scheme used is *not* strong receiver binding.
- **Client-side evidence generation.** In TLS oracles, a client is unable to prove arbitrary statements on transmitted data once a session is completed without incurring significant overhead. We introduce a protocol that enables client side sovereignty by efficiently transforming an AEAD ciphertext to a SNARK friendly commitment to the respective plaintext. Our solution does not require the client to obtain knowledge on the structure of the server response before creating a proof for the proxy to verify.
- **Evaluation and Application.** We implement ORIGO in a realistic setting with a real server [10]. For a server response of 1.4kB, the online communication required is 2.4kB — 375× less than the most performant related work relying on semi-honest 2PC. By globally dispersing clients, we show that ORIGO is less dependent on the client location, and up to 4.6× faster for remote clients. We leverage our real-world implementation and introduce a novel application by integrating ORIGO with Gitcoin Passport [5], extending it to be used with private and personal data.

Limitations. Notably, ORIGO assumes that the client cannot mount a MITM attack between the proxy verifier and the server in the TLS 1.3 session. For prover integrity, we therefore assume that the proxy can reliably connect to the server throughout the session. Although this setting assumes a weaker network adversary, we regard it as feasible due to existing detection techniques on other layers of the OSI stack [9, 57, 59, 68], and similar assumptions in

Table 1: Comparison of related works on data provenance. For communication complexity, n signifies the size of the request/response sent to/by the server.

System	Setting	Legacy Compatibility	MITM Security	Communication
DECO [67]	External	✔	✔	O(n)
DECO/Proxy	Proxy	✔	✗	O(n)
XYWY23 [63]	External	✔	✔	O(n)
Janus [46]	Proxy	✔	✔	O(n)
DiStefano [22]	External	✔	✔	O(n)
DIDO [23]	External	✔	✔	O(n)
This Work	Proxy	✔	✗	O(1)

other proxy applications [60, 67]. Our main focus is to enable improved performance, assuming that MITM between the proxy and the server is infeasible.

Another limitation in our protocol is the overhead introduced by the pre-processing required for SNARKs. In our evaluation, the preprocessing of circuits, especially to obtain the prover key, introduce a significant offline runtime overhead. Further, ORIGO inherits the randomized setup to sample a trapdoor from the respective SNARK utilized (in our case, Groth16). Plonk [33] would yield a universal and upatable SNARK. Fractal [24] would not require a trusted setup.

2 RELATED WORKS

Proof Systems for TLS. Many works use proof systems with TLS to filter network traffic [38], blindly obtain certificates [60], prove non-repudiation [21, 55], and provide off-chain data to blockchains [66, 67]. Many require server-side changes [21, 55], which are insufficient for adoption. We provide a comparison to the most relevant works as follows. TLS-N [55] is a server-side extension that allows for the generation of privacy-preserving proofs on the contents of a TLS session. As their work proposes a server-side extension, cooperation of servers in adoption of their protocol is crucial. Our protocol for evidence generation is similar to their proposed server-side extension, with the important difference that evidence is generated by the *client* in our setting, without the need for server-side adaptation. DECO[67] introduces the first TLS oracle for modern TLS versions without server-side adaptation (cf. Table 1). Initially derived from the PageSigner [2, 3] protocol first mentioned in 2014, their protocol asks the verifier and the client to jointly emulate a TLS client interacting with an unmodified server. In Appendix C.4 the authors of DECO describe an extension to their protocol which is similar to our protocol — the verifier acts as a proxy between the server and the client. However, their extension relies on 2PC during the TLS handshake in the extended protocol. Grubbs *et. al* [38] introduce *Zero Knowledge Middleboxes*, where clients send queries and simultaneously prove in zero knowledge that their traffic is compliant with a network policy. Here, the client only commits to its own request, rather than the response received from the server.

In addition to the above, there is a set of works conceived concurrently to our work. Xie *et. al* [63] introduce the *garble-then-prove* paradigm, which replaces authenticated garbled circuits and SNARKs with semi-honest garbled circuits and interactive ZKPs based on vector oblivious linear evaluation (VOLE). Their work introduces a 14× improvement in communication and up to 15× improvement in runtime as compared to DECO. They also provide a protocol for converting ciphertexts to SNARK compatible commitments. Celi *et. al* [22] introduce *DiStefano* which guarantees ring privacy by using ring signatures produced over TLS certificates and relies on maliciously secure 2PC, similar to DECO. Lauinger *et. al* [46] introduce *Janus*, a TLS oracle tailored to TLS 1.3 in the proxy setting for verifying the provenance of kilobytes of data, relying on an honest verifier ZK proof system. Janus employs handshake *and* record layer 2PC on AES encrypted counter blocks, rendering it secure against MiTM attacks in the proxy mode. Notably, all of the above utilize 2PC and hence require communication in size linear to the number of gates in the garbled circuit.

Regular Expressions in Probabilistic Proofs. A recent line of work focuses on optimizing regular expressions in probabilistic proofs, which can be utilized in TLS oracles to prove that the plaintext underlying a ciphertext fulfills a specified set fo constraints. Although this problem is not the core focus of this work, we consider it important orthogonal work. Zombie [65] extends ZKMB [38] and allows proofs on DNS requests by transforming a regular expression to an NFA and uses the sum-check based SpartanNIZK as the probabilistic proof system. Luo *et. al* introduce zk-regex [48], which follows a similar NFA approach as Zombie but transforms the NFA to a Boolean circuit and uses MPC-in-the-head for the proof. Raymond *et. al* introduce zkreg [53] for matching regular expressions based on the Aho-Corasick automaton and use a commit-and-prove scheme to prove membership in large dictionaries of strings. Out of all, Zombie supports the most expressive expressions.

TLS Security and Integrity. Other related works include TLS security analysis and message franking protocols. Message Franking protocols follow a similar setting as ORIGO, as the participants engage in a protocol that similarly requires the receiver of a message to report upon the message content [27, 39–41, 58]. However, the setting of message franking differs to our setting, as the server verifying the message reported by a message receiver does not keep track of the ciphertext initially sent by the message sender. Recent works on message franking are especially relevant, as they depict attacks that allow equivocation about its content [39].

3 PRELIMINARIES

3.1 Cryptographic Primitives in TLS

TLS is a cryptographic protocol designed to provide secure communication over a computer network. It ensures that the data transmitted between a client and a server remains confidential and cannot be intercepted or tampered with by third parties. The most recent version is TLS 1.3 [54], which improves over TLS 1.2 [26] by reducing the number of applicable algorithms for message encryption and authentication, and further introducing a 0-RTT mode, which optionally enhances performance at the cost of security. TLS 1.3 relies on AEAD, a family of encryption algorithms which simultaneously ensure confidentiality and authenticity of transmitted data. On a high level, a TLS protocol consists of two layers — a *handshake layer*, where the server and the client negotiate the traffic encrypting keys (i.e., the Client Application Traffic Key CATK and Server Application Traffic Key SATK), and a *record layer*, which applies the traffic keys in an AEAD algorithm to securely send application

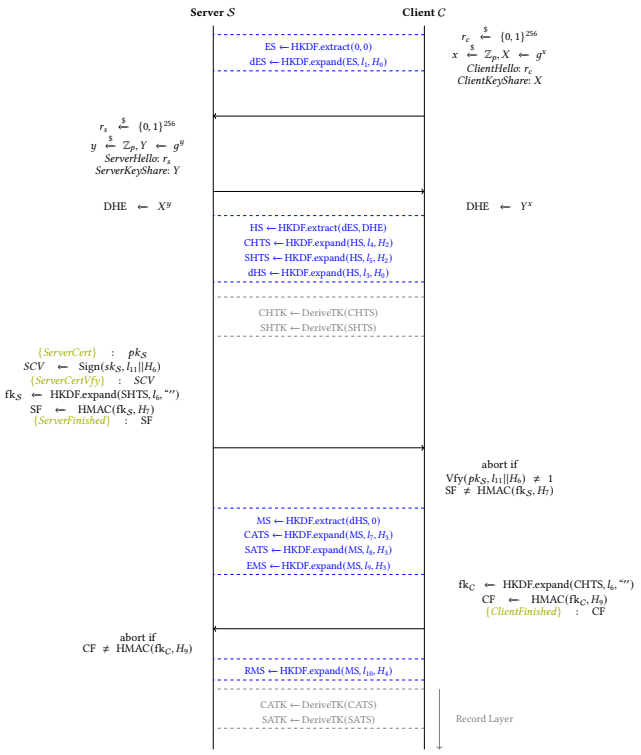

Figure 2: TLS 1.3 1-RTT handshake transcript *without* optional messages and client certificate verification. Secrets and keys derived by both server and verifier are highlighted in blue and gray respectively. Handshake messages encrypted with handshake encryption keys are highlighted in green. We refer the reader to RFC 8446 [54] and related work [29] for abbreviations and extended definitions.

data. We provide an overview of TLS 1.3 in one round-time trip (1-RTT) mode without pre-shared keys in Figure 2.

HKDF and HMAC. Key Derviation from the established DHE secret HMAC-based Extract-and-Expand Key Derivation Function (HKDF) follows the *encrypt-and-expand* paradigm, where the extraction function HKDF.extract(SALT, IKM) derives a pseudorandom key from the keying material and the expansion function HKDF.expand(PRK, INFO, L) expands the pseudorandom key into another pseudorandom key of length L. Both HKDF.extract and HKDF.expand internally make use of an Hashed Message Authentication Code (HMAC) [45], which is augmented with a hash function H (usually SHA-256). An HMAC is computed as follows:

$$\text{HMAC} = H(K \oplus \text{OPAD}, H(K \oplus \text{IPAD}, M)) \tag{1}$$

The HMAC takes as input the message M and is keyed with a key $K \in \{0,1\}^\lambda$. The security parameter λ determines padding length in the inner and outer hash [45].

AEAD. In TLS 1.3, the application data is fragmented in *records*, where each record has a maximum size of 16kB, such that one record contains a maximum of 1024 chunks of size 16 byte or 128 bit. Each

$F(K, IV, \{P_1, P_2, ..., P_m\})$
\\ Derivation of IV and Encryption
\\ $IV \leftarrow$ HKDF.expand($SATS$, "iv", l)
$R_i \leftarrow E_K\{IV||ctr_i\}$
$C_i \leftarrow R_i \oplus P_i, \forall i \in m$

\\ GHASH Tag computation
$H \leftarrow E_K\{0^{128}\}$
$X \leftarrow \sum_{i=1}^{m} C_i H^{m-i}$
$T \leftarrow E_K\{IV\} \oplus X$
Return $\{C_1, C_2, ..., C_m\}, T$

Figure 3: Description of AES-GCM as applied in the TLS 1.3 record layer. The nonce applied in AES-GCM is directly derived from the handshake with a hash-based key derivation function. For clarity, GCM is simplified as compared to the NIST specification [1].

record is encrypted and authenticated through AEAD. TLS 1.3 supports a set of ciphersuites, namely TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384 and TLS_CHACHA-20_POLY1305_SHA256, where support for the first is mandatory and the latter two are optional. Hence, the most common mode of operation for the underlying block cipher is AES in Galois Counter Mode (GCM) with an output length of 16 bytes or 128 bit. Importantly, the traffic sent by the client and server in the record layer are encrypted with their respective symmetric keys CATK and SATK. Both keys are known by the client and server. In the remainder of this work, we assume AES-GCM is applied for both encryption and decryption, where the ciphertext carries a 128-bit authentication tag. We detail a simplified description of AES-GCM as applied in TLS 1.3 to encrypt record layer data in Figure 3. For simplicity, we ignore associated data. $F(K, IV, \{P_1, P_2, ..., P_m\})$ uses the key K with the AES blockcipher E of block size 128 bit. It encrypts the plaintext chunks $\{P_1, P_2, ..., P_m\}$, where m is maximally 1024, and outputs the Tag T and the chunks of ciphertext $\{C_1, C_2, ..., C_m\}$. The tag is computed with an $\epsilon - AXU$ hash function that relies on binary polynomials of degree 128 in the Galois Field $GF(2^{128})$. The polynomial is evaluated at the encryption of the 0-vector H and successively XOR-ed with the encryption of the initialization vector IV to obtain the tag T (for details, see the GHASH Function [1]).

3.2 zkSNARKs

In the realm of cryptographic protocols, zero-knowledge proofs (ZKPs) enable one party, referred to as the prover, to convince another, the verifier, of the veracity of a statement, while not disclosing any other information aside from its validity. At its core, the objective of a ZKP is to establish that a given *instance* x and its associated *witness* w conform to an NP relation $\mathcal{R}$. This can be formally depicted by the relationship $(x, w) \in \mathcal{R}$, which can be ascertained in polynomial time by a Turing machine.

A typical ZKP encompasses three algorithms:

- Setup(pp) $\rightarrow$ (pk, vk). Given public parameters pp as input, compute and output proving and verification keys pk and vk, respectively.

$\text{sr} - \text{BIND}^{\mathcal{A}}_{\text{AEAD}}$
$((K, IV, P), (K', IV', P'), C, T) \leftarrow \mathcal{A}$
$b \leftarrow \text{Ver}(K, IV, P, C, T)$
$b' \leftarrow \text{Ver}(K', IV', P', C, T)$
if $(K, IV, P) == (K', IV', P')$:
    return false
return $(b = b' = 1)$

Figure 4: Strong Receiver Binding Security Game for an AEAD algorithm. An adversary wins if it finds a differing key K, nonce IV and plaintext P for a fixed ciphertext C and authentication tag T.

- $\text{Prove}(\text{pk}, x, w) \rightarrow \pi$. Given the proving key pk, the instance x, and the witness w, such that $(x, w) \in \mathcal{R}$, as input, compute and output a proof π.
- $\text{Verify}(\text{vk}, x, \pi) \rightarrow 0/1$. Given the verification key vk, the instance x, and the proof π as input, output 1 if the proof is valid and 0 otherwise.

We say that (Setup, Prove, Verify) is a zkSNARK if it satisfies completeness, succinctness, knowledge soundness and zero-knowledge. Completeness ensures that for any genuine statement, an honest prover should be able to convince an honest verifier. Knowledge soundness, informally, describes that for every adversarial prover, there exists an *extractor*, such that whenever the prover convinces the verifier to output 1, the extractor can use the prover algorithm to output w such that $(x, w) \in \mathcal{R}$ [15]. Further, a SNARK is succinct if the size of the proof and the verification time are $o(|R|)$.

3.3 Security Guarantees & Threat Model

Throughout this paper, we assume that the adversary can arbitrarily deviate from the specified protocols. The server is assumed to behave correctly and respond honestly to any request sent by a client. Due to the impossibility of preventing a MITM attack in the proxy setting, we assume that an adversarial client cannot perform a MITM attack between the server and the proxy. We assume that security, as proven in numerous previous works [32], holds for the standard TLS protocol in version 1.3. The security properties to be retained by a protocol proposing a TLS oracle are the following:

Client Integrity. A malicious client cannot convince a verifier of an untrue statement about the TLS session.

Verifier Integrity. A malicious verifier cannot prevent a client from receiving the correct message from the server.

Privacy. A malicious proxy verifier cannot obtain any knowledge beyond the size of the message that it forwards.

4 ATTACK ON TLS 1.3 IN THE PROXY SETTING

In this section, we detail an attack on the integrity of messages sent via TLS 1.3. We show that the receiver of a ciphertext $\{C_1, C_2, ..., C_m\}$ (i.e. encrypted TLS record) can equivocate about the underlying plaintext $\tilde{P} = \{P_1, P_2, ..., P_m\}$. Grubbs *et. al* detail an attack for GCM that allows a receiver to equivocate about the underlying plaintext by arbitrarily choosing the ciphertext $\{C_1, C_2, ..., C_m\}$ in the context of "Message Franking" [39], where the receiver of a message intends to report the content of a received message to a third party. The third party needs to be sure that the reported message is the one initially sent by the sender. It is hence shown that GCM is not *receiver binding* due to the the applied Carter-Wegman MAC not being collision resistant [39]. We introduce an orthogonal notion called *strong receiver binding*, which further restricts the adversary in manipulating the ciphertext and the associated MAC. This is important in the proxy setting of a TLS oracle, as in this case the proxy obtains a ciphertext from a server, which is assumed to behave honestly. If the client can find a colliding ciphertext for a differing plaintext, additional care needs to be taken to ensure the integrity of the content with respect to verification by the proxy.

Integrity Attack on TLS 1.3. Trivially, AES-GCM in TLS 1.3 only provides ciphertext integrity between the two participating entities, the sender and the receiver, due to symmetric keys that allow the receiver (i.e., the client) to encrypt and authenticate arbitrary ciphertexts [67]. We further show in the following that AES-GCM as applied in TLS 1.3 is not only not receiver binding, but also not strong receiver binding. To do so, we formalize the pseudocode of the security game $\text{sr} - \text{BIND}^{\mathcal{A}}_{\text{AEAD}}$ in Figure 4. The adversary $\mathcal{A}$ has to output a pair of triples specifying the symmetric key, nonce and authentication tag. The adversary wins if verification succeeds for both tuples, given the same ciphertext and tag. The formal advantage of an adversary $\mathcal{A}$ can be specified as

$$\text{Adv}^{\text{sr-BIND}}(\mathcal{A}) = \Pr[\text{sr-BIND}^{\mathcal{A}}_{\text{AEAD}} \rightarrow \texttt{true}]$$

For our specification, we rely on the definition of committing AEAD as introduced by Grubbs *et. al* [39]. Note that sr-BIND (strong receiver binding) security implies r-BIND (receiver binding) security.

We give an adversary $\mathcal{A}_{GCM}$ which wins if it outputs $((\tilde{P}, IV, K), (\tilde{P}', IV', K'), \{C_1, C_2, ..., C_m\}, T)$, such that

$$F(K, IV, \tilde{P}) = F(K', IV', \tilde{P}')$$

Choosing $(K', IV', \tilde{P}')$ is trivial, as the adversary must solve the following equation in order to obtain a conflicting decryption to the ciphertext $\{C_1, C_2, ..., C_m\}$ and authentication tag T:

$$E_K\{IV\} + E_{K'}\{IV'\} + \sum_{j=1}^{m}(R_i \oplus P_i)H^{m-i}$$
$$+ \sum_{j=1}^{m}(R'_i \oplus P'_i)H^{m-i} = 0$$

To compute a valid Tag T, $\mathcal{A}_{GCM}$ chooses a key K', computes X and chooses IV' such that $T = T'$. To successively compute $\tilde{C}$ and $\tilde{C}'$, such that $\tilde{C} = \tilde{C}'$, $\mathcal{A}_{GCM}$ computes the series R'_i from IV' and computes $\tilde{P}$ such that the above equation is fulfilled. We elaborate on the practical feasibility of this attack in Appendix C.

Why receiver-binding holds in TLS 1.2. In TLS 1.2, the IV for AEAD cipher suites is generated using a pseudorandom function (PRF) based on the master secret and a nonce value. The nonce value is generated by the sender and included in the record header of the TLS message. The IV is derived by applying the PRF to the master secret and the nonce value and using the first N bytes of the output as the IV, where N is the size of the IV required for the particular AEAD cipher suite being used. In TLS 1.3, the IV is generated using a key derivation function (KDF) based on the

***Circuit* — Key Derivation:** (HS; $\mathbf{H}_2$, $\mathbf{H}_3$, **SHTS**)

1. **SHTS**← HKDF.expand (HS,"s hs traffic" || $\mathbf{H}_2$)
2. dHS ← HKDF.expand (HS,"derived", H(" "))
3. MS ← HKDF.extract (dHS, 0)
4. CATS ← HKDF.expand (MS, "c ap traffic" || $\mathbf{H}_3$)
5. SATS ← HKDF.expand (MS, "s ap traffic" || $\mathbf{H}_3$)
6. CATK← DeriveTK(CATS)
7. SATK← DeriveTK(SATS)

Figure 5: ZKP circuit to prove the TLS 1.3 key derivation. We highlight the public input with bold text.

secret and the record sequence number, and it is used to initialize the AEAD cipher. The secret is a value derived from the master secret and the handshake hash, and the record sequence number is an incrementing counter that is used to differentiate each record in a TLS session. The IV is not transmitted in the header of the TLS record. Instead, it is used internally by the AEAD cipher to encrypt and decrypt the message payload. Consequently, the recipient of a message over TLS 1.3 can arbitrarily choose the IV when reporting to a third party. We ensure that this attack is not possible in our protocol tailored to TLS 1.3 by deriving the *IV* in a SNARK and proving correct encryption with the correct application traffic key.

5 THE ORIGO PROTOCOL

In this section we describe ORIGO, a novel TLS oracle in the proxy setting. The protocol operated in three phases. Whereas the first phase involves the server, proxy verifier and the client, the second and last phase only involve the latter two. The last phase is the only phase involving online operations beyond the TLS session.

- **Handshake & Request Execution.** In this phase, the client engages in the handshake with the server, sends a request and obtains a response which is fragmented into records of max. 16kB size. The handshake, request and response is routed through the proxy. The proxy obtains the encrypted TLS 1.3 handshake transcript, the encrypted query and the encrypted server response.
- **Pre-Processing.** During pre-processing, the client generates public values from the handshake transcript for optimized proof generation.
- **Proof Generation & Verification.** The client utilizes the pre-processed private and public input to generate a proof for integrity of data transmitted in the TLS session. The pre-processed public values and the proof are transmitted to the verifier, which verifies the authenticity of the handshake by verifying the server certificate, assigning the public input to the arithmetic circuit and verifying data authenticity.

Following, we detail *(i)* the key derivation circuit, *(ii)* the formal specification of ORIGO, Π_{ORIGO}, and *(iii)* the extension for client-side evidence generation. In Appendix E we show that Π_{ORIGO} securely realizes the ideal functionality for data provenance $\mathcal{F}_{\text{DP}}$.

5.1 Proof of Key Derivation

The key derivation circuit is a building block used by the proxy to force the client into computing correct and non-ambiguous proofs of data provenance. First, the proxy must verify a valid authentication of the intercepted TLS 1.3 transcript data. In order to authenticate the transcript data, the proxy uses the server's Public Key Infrastructure (PKI) certificate to validate the server's transcript signature. Second, the proxy requires the client to show a non-ambiguous mapping of private TLS 1.3 session keys against intercepted and public TLS 1.3 transcript data. Correctness of data provenance holds because the challenges ensure that the client cannot forge the signature of transcript data. Thus, under the assumption of an honest server, the transcript data is correct. Further, non-ambiguity of the session keys lets the proxy verify that the record data complies with a correct TLS 1.3 session.

To verify non-ambiguity and, with that, correctness of session keys, the proxy demands the client to compute a ZKP circuit. The ZKP circuit ensures integrity of a cryptographically binding mapping between private session keys and public TLS 1.3 transcript data. A cryptographically binding computation is a collision resistant function evaluation which guarantees an unequivocal mapping of input data to a specific output. Further, the ZKP circuits maintains privacy of private session keys. If the client is able to compute a valid proof of the ZKP circuit, the proxy is convinced that non-ambiguity of session keys holds.

The naive approach to compute a cryptographically binding mapping between private session keys and their public transcript is to follow TLS 1.3 key exchange and key derivation specification. In the TLS 1.3 handshake, the server and client randomly choose secrets $x \xleftarrow{\$} \{0,1\}^{256}$ and $y \xleftarrow{\$} \{0,1\}^{256}$, which they exchange based on public diffie-hellman key exchange parameters $\mathbf{X}{=}g^x$ and $\mathbf{Y}{=}g^y$. The parties involved in the secret exchange obtain a shared secret by computing DHE $\leftarrow \mathbf{Y}^x = \mathbf{X}^y$ with their respective secret randomness. With access to the shared secret DHE, server and client are eligible to compute the handshake secret HS ← HKDF.extract (dES, DHE), which only they can compute by knowing DHE. The parameter dES is publicly known and computed as dES ← HKDF.expand (ES,"derived", H(" ")) with ES ← HKDF.extract (0,0). With access to HS, the client and server derive handshake and record layer application traffic secrets. Grubbs *et.al*[38] show that deriving HS based on the private input y and public input $\mathbf{Y}$ leads to a non-ambiguous sample of HS. Further, derivation and verification of the server certificate signature must be computed in the ZKP circuit to verify a correct authentication of $\mathbf{Y}$ and, thus, HS. Doing all these computation in the ZKP circuit is costly. Luckily, a shortcut exists.

Due to the key independence property of TLS 1.3 [28], the client can disclose the Server Handshake Traffic Secret (SHTS) to the proxy without compromising security of HS and record layer application traffic secrets. Leaking SHTS is possible because HKDF.expand utilizes a one-way collision resistant hash function HS with sufficient entropy of the input secret. The proxy utilizes SHTS to derive the handshake encrypting key SHTK to decrypt handshake traffic and verify the server finished message (cf. SHTS verification in Figure 6). Further, the proxy accesses the server's certificate and can efficiently verify transcript authenticity in a local out-of-circuit computations. We depict the efficient key derivation circuit in Figure 5, where we follow the convention of notations introduced by Dowling *et. al* [29]. Notice that transcripts hashes $\mathbf{H}_7$ = H(ClientHello‖…‖ServerCertVfy), $\mathbf{H}_2$ = H(ClientHello‖ServerHello),

Π_{Auth}(**SHTS**, $\mathbf{H}_7$, **SF**)
1. $\text{fk}_{\mathcal{S}} \leftarrow$ HKDF.expand (**SHTS**, "finished" || " ")
2. SF' $\leftarrow$ HMAC($\text{fk}_{\mathcal{S}}$, $\mathbf{H}_7$)
3. SF' $\stackrel{?}{=}$ **SF**
4. ok $\stackrel{?}{=}$ verifyCertificate()

Figure 6: Protocol for out of circuit verification of the server-side certificate to ensure authenticity of the obtained ciphertext and handshake transcript.

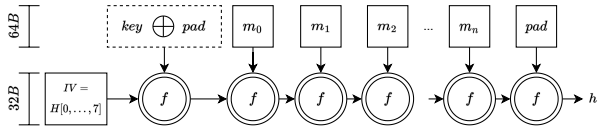

Figure 7: Circuit optimization based on Merkle-Damgård structure of SHA-256. Dotted boxes indicate private input, whereas solid boxes indicate public input. For a private *key*, the prover computes the first f in-circuit. All remaining intermediate hash values f can be computed out of circuit and compared with the public hash h.

$\mathbf{H}_3$ = H(ClientHello||... ||ServerFinished), and $\mathbf{H}_0$ = H(" ") are computed at the proxy. Further, application traffic secrets are computed according to the formula SATK| CATK= (key,iv) = DerivETK(secret) = (HKDF.expand (secret,"key",H(" "), L_k), HKDF.expand (secret,"iv",H(" "), L_{iv}))), where L_k/L_{iv} indicate the key or iv length of the selected TLS 1.3 cipher suite. Following, we show how the Key Derivation circuit can be further optimized.

HMAC Optimization. The TLS 1.3 key derivation can be further optimized when looking at the structure of HMAC with SHA-256. The optimization is relevant because HKDF.extract and HKDF.expand both make use of HMAC (see Section 3). Further depending on the required key size of the TLS 1.3 cipher suite, HKDF.expand calls HMAC multiple times. For example, with `TLS_AES_128_GCM_SHA256` 128 bit encryption keys are used such that both functions HKDF.extract and HKDF.expand call HMAC once internally.

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. The input to the inner hash function is of size 64 + 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 [47]. 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 [67]. As a result, the output of the compressing blockcipher in SHA256 can be used as public input to reduce ZKP circuit complexity.

Figure 7 shows 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 f if the message input m is publicly known.

Detailed Invocation of Compression Functions. Figure 8 shows the computation trace of the TLS 1.3 key derivation, detailed to the level of single invocations of the compression function f. The figure summarizes computations executed at the proxy as the verifier (v), and the client as the prover (p, zk). "zk" indicates computations that are part of the optimized ZKP circuit. Public parameters are highlighted in bold. Note that dHS^{in} is the only private input on the client side. The ZKP circuit shows that if the client discloses $\mathbf{h}^{\text{HS},opad}$ and $\mathbf{SHTS}^{\text{in}}$ as public input (line 2 and 3), SHTS (line 4) can be computed out of circuit to verify SF, and dHS (line 11) can be computed in-circuit, mapping to the only private input dHS^{in} of the optimized ZKP circuit. Thus, deriving either one of the application traffic keys CATK or SATK requires 8 in-circuit invoations of the SHACAL-2 compression function f (lines 11, 14, 17, 20, 21).

5.2 Formal Protocol Specification

We formally specify our full protocol Π_{ORIGO} in Figure 10. Π_{ORIGO} relies on $\mathcal{F}_{ZK}$ to abstract away the complexity of ZKPs in the protocol, and hence operates in the $\mathcal{F}_{ZK}$-hybrid model. Π_{ORIGO} is parametrized by a signature scheme $\Sigma(KGen, Sig, Vf)$ and an AEAD algorithm $\mathcal{AE}(Enc, Dec)$, defined by the ciphersuite (TLS_AES_128_GCM_SHA256 & Curve P-256).

Notation. In our formal specification, we denote the TLS server as $\mathcal{S}$, the proxy verifier as $\mathcal{V}$ and the TLS client as C. In ORIGO, the verifier $\mathcal{V}$ acts as a Proxy between C and $\mathcal{S}$. $\Pi_{\mathcal{S}}$ denotes the unmodified server protocol as specified in TLS 1.3. We further denote the handshake transcript between $\mathcal{S}$ and C as Γ. The encrypted TLS request $\hat{Q}$ and response $\hat{M}$ are sent by C and $\mathcal{S}$ respectively. We denote a request response pair as a tuple $\langle \hat{Q}, \hat{M} \rangle$. In general, we denote encrypted values with a caret, e.g., $\hat{M}$ for M.

a) Handshake. To send a request, the client C first has to engage in a TLS 1.3 handshake with the server $\mathcal{S}$ by initializing the protocol. As discussed in Section 3 TLS handshake we assume in our protocol is standard, besides that each message from the handshake is routed through $\mathcal{V}$, such that $\mathcal{V}$ obtains the handshake transcript $\hat{\Gamma}$, which is encrypted after establishment of the handshake traffic keys (SHTK and CHTK in Figure 2).

b) Request Execution. Upon establishment of the traffic encrypting keys, C is ready to send a request $\hat{Q}$ to $\mathcal{S}$. The client C builds the request by invoking "request". The client embeds the secret for authentication with the server $\theta_{\mathcal{S}}$ (e.g., API key) and encrypts the request with the client application traffic key CATK. The client successively calls the "request" subroutine of the verifier $\mathcal{V}$ with the request $\hat{Q}$. The verifier $\mathcal{V}$ forwards $\hat{Q}$ to $\mathcal{S}$ and, under the assumption that $\mathcal{S}$ operates honestly, receives a record $\hat{R}$ from $\mathcal{S}$. The proxy stores $\hat{R}$ and forwards the record to C.

Initialize:
l_2 = "tls13 s hs traffic"; l_3 = "tls13 s ap traffic"
l_0 = "tls13 derived"; l_f = "tls13 finished"
l_k = "tls13 key"; l_{iv} = "tls13 iv";
$m^{\mathbf{H}_2} = 32 \,||\, len(l_2) \,||\, l_2 \,||\, len(\mathbf{H}_2) \,||\, \mathbf{H}_2 \,||\, 1$;
$m^{\mathbf{H}_3} = 32 \,||\, len(l_3) \,||\, l_3 \,||\, len(\mathbf{H}_3) \,||\, \mathbf{H}_3 \,||\, 1$;
$m^{\mathbf{H}_0} = 32 \,||\, len(l_0) \,||\, l_0 \,||\, len(\mathbf{H}_0) \,||\, \mathbf{H}_0 \,||\, 1$;
$m_f = 32 \,||\, len(l_f) \,||\, l_f \,||\, 0 \,||\, 1$;
$m_k = 16 \,||\, len(l_k) \,||\, l_k \,||\, 0 \,||\, 1$;
$m_{iv} = 12 \,||\, len(l_{iv}) \,||\, l_{iv} \,||\, 0 \,||\, 1$;
Key Derivation Trace:
1. p: $h^{\text{HS},ipad}, l_x = f$ (IV, 0, HS $\oplus$ *ipad*)
2. p: $\mathbf{SHTS}^{\text{in}}, \_ = f$ ($h^{\text{HS},ipad}$, l_x, $m^{\mathbf{H}_2}$)
3. p: $\mathbf{h}^{\text{HS},opad}, l = f$ (IV, 0, HS $\oplus$ *opad*)
4. v: SHTS , $\_ = f$ ($\mathbf{h}^{\text{HS},opad}$, l, $\mathbf{SHTS}^{\text{in}}$)
5. v: $\text{fk}_{\mathcal{S}}{}^{in}, \_ = f$ (f (IV, 0, SHTS $\oplus$ *ipad*), m_f)
6. v: $\text{fk}_{\mathcal{S}}$, $\_ = f$ (f (IV, 0, SHTS $\oplus$ *opad*), $\text{fk}_{\mathcal{S}}{}^{in}$)
7. v: $\text{SF'}^{in}, \_ = f$ (f (IV, 0, $\text{fk}_{\mathcal{S}} \oplus$ *ipad*), $\mathbf{H}_7 \,||\, 1$)
8. v: SF' , $\_ = f$ (f (IV, 0, $\text{fk}_{\mathcal{S}} \oplus$ *opad*), SF'^{in})
9. v: SF' $\stackrel{?}{=}$ **SF**

10. p: $\text{dHS}^{in}, l = f$ ($h^{\text{HS},ipad}$, l_x, $m^{\mathbf{H}_0}$)
11. zk: dHS , $\_ = f$ ($\mathbf{h}^{\text{HS},opad}$, l, dHS^{in})
12. p: $\mathbf{h}^{\text{dHS},ipad}, l = f$ (IV, 0, dHS $\oplus$ *ipad*)
13. v: $\mathbf{MS}^{in}, \_ = f$ ($\mathbf{h}^{\text{dHS},ipad}$, l, 0bytes)
14. zk: MS , $\_ = f$ (f (IV, 0, dHS $\oplus$ *opad*), $\mathbf{MS}^{in}$)
15. p: $\mathbf{h}^{\text{MS},ipad}, l = f$ (IV, 0, MS $\oplus$ *ipad*)
16. v: $\mathbf{SATS}^{in}, \_ = f$ ($\mathbf{h}^{\text{MS},ipad}$, l, $m^{\mathbf{H}_3}$)
17. zk: SATS , $\_ = f$ (f (IV, 0, MS $\oplus$ *opad*), $\mathbf{SATS}^{in}$)

18. p: $\mathbf{h}^{\text{SATS},ipad}, l = f$ (IV, 0, SATS $\oplus$ *ipad*)
19. v: $\mathbf{SATK}^{in}, \_ = f$ ($\mathbf{h}^{\text{SATS},ipad}$, l, m_k)
20. zk: $h^{\text{SATS},opad}, l = f$ (IV, 0, SATS $\oplus$ *opad*)
21. zk: SATK, $\_ = f$ ($h^{\text{SATS},opad}$, l, $\mathbf{SATK}^{in}$)[:16]
22. v: $\mathbf{sIV}^{in}, \_ = f$ ($\mathbf{h}^{\text{SATS},ipad}$, l, m_{iv})
23. p: **sIV**, $\_ = f$ ($h^{\text{SATS},opad}$, l, $\mathbf{sIV}^{in}$)[:12]

Figure 8: Computation trace of the key derivation of a single TLS 1.3 application traffic key. The figure differentiates between in-circuit and out of circuit computations which are separately computed by the prover and verifier respectively. The function f is the one-way compression blockcipher SHACAL-2 of SHA256 and takes a 32 byte input as the first argument, a padding length helper as the second argument, and a 64 byte input as the third argument. The function returns the length l (required for padding) of the hashed input and the output of the SHACAL-2 blockcipher. The initialization vector of f computes as IV = $H[0, \ldots, 7]$.

c) Pre-Processing. Upon invocation of "response", the client C decrypts the record to obtain the plaintext record R. The client proceeds to pre-processes the record by finding the indices of AES blocks which match the provided ctx by invoking Π_{post} (see Figure 9). If the record in question matches the context, C outputs the pre-processed transcript Γ'. The pre-processed values in Γ' are sent

$\Pi_{\text{postR}}(R, \text{ctx})$
firstBlock $\leftarrow 0$, lastBlock $\leftarrow 0$, offset $\leftarrow 0$, found $\leftarrow 0$
for $i \leftarrow 1$ to n:
  if ctx[0] $\in B_i$:
    firstBlock $\leftarrow B_i$; offset $\leftarrow B_i$[ctx[0]] $- 16 \cdot i$; found $\leftarrow 1$
  if ctx["."] $\in B_i$:
    pos $\leftarrow B_i$[ctx["."]]
  if ctx[|ctx|] $\in B_i$:
    lastBlock $\leftarrow B_i$
set $\text{p}_{\text{R}} \leftarrow$ (firstBlock, lastBlock, offset, pos)
return p_{R}

Figure 9: Protocol Π_{postR} for post-processing the record to identify the correct indices of AES Blocks containing the context. Gray values identify floating point values.

to $\mathcal{V}$ in plain and decrease the number of HMAC evaluations in the Key Derivation Circuit. The secret input to the ZKP is dHS^{in}, as specified in line 11 of 8, which depicts all computations for key derivation, including circuit optimizations introduced in Section 5.1.

d) Proof Generation & Verification. To prove data provenance, C needs to prove the integrity of the plaintext underlying the obtained TLS record by *(i)* proving the key derivation (cf. Figure 8), *(ii)* proving that the authentication tag is correct, and *(iii)* proving that the private input R encrypts to the encrypted response $\hat{R}$ stored by the verifier $\mathcal{V}$. Additionally, the client proves that the TLS record in question satisfies the statement as defined by the context ctx. Formally, we adopt the ZKP functionality $\mathcal{F}_{ZK}$ as described in [61]. In $\mathcal{F}_{ZK}$, C and $\mathcal{V}$ provide their private and public inputs, and $\mathcal{F}_{ZK}$ sends *true* to $\mathcal{V}$ if the private witness provided by C satisfies the circuit C. The client utilizes $\mathcal{F}_{ZK}$ by assigning the private input w, which includes the server application traffic key SATK, the authentication secret $\theta_{\mathcal{S}}$, dHS^{in} and the plaintext record R. The public input x is assigned accordingly, and the client C invokes "prove" in $\mathcal{F}_{ZK}$. The client sends the post-processed transcript and the post-processed record by calling "proof" at the verifier $\mathcal{V}$.

The verifier parses the pre-processed transcript Γ' and obtains the public input $\mathbf{SHTS}^{\text{in}}$, through which it can derive the handshake encrypting key SHTK. Successively, the verifier $\mathcal{V}$ decrypts the encrypted handshake. The client utilizes Π_{Auth} as a subroutine to verify the server-side certificate and check the validity of SF. Upon successful verification, $\mathcal{V}$ sets as public input the post-processed values p_{R} obtained through the context, the post-processed transcript Γ' and the stored encrypted record $\hat{R}$. The verifier $\mathcal{V}$ calls "verify" in $\mathcal{F}_{ZK}$, and attests with a signature once $\mathcal{F}_{ZK}$ returns.

We characterize the security of Π_{ORIGO} through the following theorem (formal proof in Appendix E):

Theorem 1. *Assuming ECDSA is EUF-CMA secure and the compression function f is a random oracle, Π_{ORIGO} UC-realizes the ideal functionality for data provenance ($\mathcal{F}_{DP}$) in the ($\mathcal{F}_{ZK}$)-hybrid world.*

5.3 Client Side Evidence Generation

In this section, we outline how ORIGO can be extended beyond simple attestations to generate a commitment to the plaintext records

$\Pi_{ORIGO}\{\Sigma, \mathcal{AE}, \mathcal{S}, \mathcal{V}, C\}$

A client C interacts with a verifier $\mathcal{V}$ to obtain data from a server $\mathcal{S}$. The server $\mathcal{S}$ inputs a secret key $sk_{\mathcal{S}}$ and a certificate $\delta_{\mathcal{S}}$ to authenticate the data sent. The context ctx describes the key/value pair of interest. Further, the protcol is parametrized with a circuit C.

Server $\mathcal{S}$: Follow standard TLS 1.3 with verification of client certificates.

Client C:

Initialize:

Send *ClientHello* $:= r_c \xleftarrow{\$} \{0,1\}^{256}$ to $\mathcal{S}$, receive Γ from $\mathcal{S}$,

obtain CHTS, CATK, SHTS, SATK and verify the server certificate $\delta_{\mathcal{S}}$.

On input (sid, "request", $\theta_{\mathcal{S}}$) from environment $\mathcal{Z}$:

set Q with $\theta_{\mathcal{S}}$

set $\hat{Q} := \mathcal{AE}.Enc(Q, \text{CATK})$

send (sid, "request", $\hat{Q}$) to $\mathcal{V}$

On receive (sid, "response", $\hat{R}$) from environment $\mathcal{Z}$:

set $R := \mathcal{AE}.Dec(\hat{R}, \text{SATK})$

run $\Pi_{\text{postR}}(R, \text{ctx})$ locally to obtain p_R

run $\Pi_{\text{postH}}(\Gamma)$ to obtain Γ'

extract dHS^{in} and set $w := (\text{SATK}, \theta_{\mathcal{S}}, \text{dHS}^{in}, R)$

set $x := (\text{p}_\text{R}, \Gamma', \hat{R})$

send (sid, "prove", C, x, w) to $\mathcal{F}_{ZK}$

send (sid, "proof", Γ', p_R) to $\mathcal{V}$

Proxy $\mathcal{V}$:

Initiatlize:

Initialize $(pk_{\mathcal{V}}, sk_{\mathcal{V}}) \leftarrow \Sigma.KGen(1^\lambda)$,

transcribe encrypted handshake transcript $\hat{\Gamma}$.

On receive (sid, "request", $\hat{Q}$) from C:

store $\hat{Q}$ and send (sid, "request", $\hat{Q}$) to $\mathcal{S}$

On receive (sid, "response", $\hat{R}$) from $\mathcal{S}$:

store $\hat{R}$ and send (sid, "response", $\hat{R}$) to C

On receive (sid, "proof", Γ', p_R) from C:

Parse Γ', derive SHTK from $\textbf{SHTS}^{\text{in}}$

set $\Gamma := \mathcal{AE}.Dec(\hat{\Gamma}, (\text{SHTK}))$

invoke Π_{Auth} and verify the server certificate $\delta_{\mathcal{S}}$

set $x := (\text{p}_\text{R}, \Gamma', \hat{R})$

send (sid, "verify", C, x) to $\mathcal{F}_{ZK}$

wait for $\mathcal{F}_{ZK}$ to respond, if *false* return $\perp$

send $\sigma := \Sigma.sign(\text{ctx}, \hat{R})$ to C

Figure 10: The ORIGO post-handshake protocol specification. The protocol is parametrized by a signature scheme $\Sigma(KGen, Sig, Vf)$ and an AEAD scheme $\mathcal{AE}(Enc, Dec)$, which is specified by the ciphersuite applied in TLS 1.3.

transmitted in a TLS session. The rationale of evidence generation follows the paradigm of evidence generation as introduced by Ritzdorf *et. al* [55], with the important difference that evidence is generated by the *client* rather than the *server*. This preserves that no server-side changes are demanded to generate evidence of data provenance. In Figure 11 we depict the process of client side evidence generation for 4 plaintext chunks of size 16B. We realize the conversion of plaintext to hiding and binding commitments in two steps. First, the client expands the per-record nonce into a salt tree. Second, the client generates a salted Merkle Tree, where the tree leafs are instantiated as hash functions over the plaintext AEAD blocks salted with the per-block salts generated by the salt tree. We assume that a hash function salted with a sufficiently large salt therein provides a hiding and binding commitment. Importantly, the initial nonce is the initialization vector IV, which is derived from SATS in TLS 1.3 and directly binds the nonce utilized to generate salt values to the TLS handshake.

Generating a Proof of Evidence. To generate evidence, the client needs to convince the proxy verifier that the root hash of the salted merkle tree h_0 was correctly derived with respect to the salt tree expansion and Merkle tree computation. Therefore, the client computes a ZKP, attesting the correct derivation of the root hash h_0, in addition to the proof of key derivation, authentication tag and encryption of records as described in Section 5.2. As in TLS 1.3, we instantiate the expansion function that is used in the salt tree with HKDF.expand, which is a common pseudorandom function for deriving keying material that is cryptographically secure, contextually bound, and properly domain-separated. However, due to the requirement of computing HKDF.expand inside a SNARK, and the omitted demand to base HMAC on SHA-256, the MiMC hash [12] can be used in HKDF.expand for HMAC. The output size of a MiMC hash is determined by the field that it operates over, as it is designed to work in a finite field setting. In our implementation, we instantiate a SNARK over BN254, which operates over a prime field of 254 bit. This is differing to SHA-256, which outputs a 256 Bit hash value. Therefore, each salt in the leafs of the salt tree is of size 127 bit. As the maximum number of blocks in TLS record is 1024, the maximum depth of the salt expansion tree is 10, which results in a maximum of 1023 invocations of HKDF.expand. When instantiated with MiMC, this is equivalent to 1022 HMAC evaluations with a pseudorandom key of 127 bit and a single HMAC evaluation with a pseudorandom key of 96 bit (the initial IV). As each HMAC internally composes two hash evaluations (cf. Section 3), this demands 2046 MiMC invocations over BN254.

The same optimizations for proof generation as in the key derivation circuit can be applied for the generation of the salt and merkle tree, such that the number of MiMC evaluations can be divided by two, as described in Section 5.1. Additionally, recent work by Belling *et. al* [16] describes how hashing in a SNARK can be significantly sped up by embedding the GKR [36] verification algorithm in a Groth16 prover. We evaluate the performance of client side evidence generation and optimizations in Section 7.2.

Presenting Evidence to a Third Party. The client submits h_0 as an additional public input to the verifier, such that $x := (\text{p}_\text{R}, \Gamma', \hat{R}, h_0)$ (cf. Figure 10). The private input remains $w := (\text{SATK}, \theta_{\mathcal{S}}, \text{dHS}^{in}, R)$. The verifier derives the SATK, and additionally derives the salt and computes the salted merkle tree to check the validity of the provided Merkle Root. Once verified, the prover obtains a signature from the verifier which attests to the validity of the provided h_0. If the client intends to disclose a full block in plain, it can open the commitment to the plaintext by disclosing a number of hashes and salts that is logarithmic in the number of elements in the salt and merkle tree. An example of disclosing P_3, P_4 to a third party in plain can be observed in Figure 11, where all disclosed values are highlighted in green. Similarly, the client can prove an arbitrary statement on any non-disclosed block by proving ZK-PoK$\{s_i, P_i : h_0 = MT(H_{M,i}(s_i, P_i))\}$ and disclosing all other outputs of $H_{M,j}$ where $j \neq i$. This is especially interesting, as the client does not need to decide on the statement or context to be attested by the proxy verifier. Instead, the proxy verifier can simply authenticate the evidence provided by the client, and the client can reveal a custom amount of information whenever demanded once the TLS session has concluded.

Further Optimization. To optimize the performance of client side evidence generation, one may resort to an alternative commitment based on the counter block $\text{CB}_i = \text{E}_k(IV||0||i+1)$. This

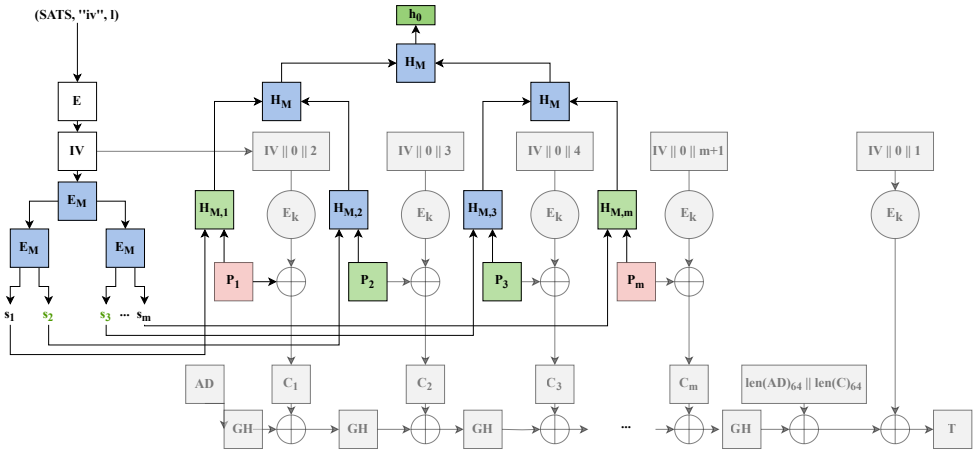

Figure 11: Client side evidence generation for a single TLS record. The example shows 4 blocks encrypted with AES-GCM (components of AES-GCM are highlighted in light gray). The maximum number of blocks m is 1024, where each block has a size of 16B. E is the expansion function HKDF.expand instantiated with SHA-256, whereas E_M is HKDF.expand instantiated with MiMC. To disclose P_2, P_3 to a third party, the client reveals all values highlighted in green. Additionally, the third party verifies the signature of the proxy verifier, which verifies that h_0 is authentic.

is possible as the AES encryption of the counter values in GCM mode act as a pseudorandom function. As long as the counter values are unique for each block of plaintext under a given key, the resulting keystream will be pseudorandom, making the ciphertext appear random and ensuring that repeated plaintexts do not result in repeated ciphertexts. Hence, a client can utilize CB_i as a sufficiently random value to build the Merkle Tree obtain the hash h_0. This demands for additional storage of counter blocks by the client, in addition to the plaintext and ciphertext. After proving correct derivation of the counter block Merkle Tree, the client can prove an arbitrary statement on any non-disclosed block by computing ZK-PoK$\{k : h_0 = MT(H_{M,i}(\text{CB}_i))\}$, where k is SATK for the server response, or CATK for the request. This introduces a significant performance improvement, as it does not demand the generation of any salt values, rendering the salt tree expansion superfluous.

6 IMPLEMENTATION

We implement ORIGO in $\sim 5k$ lines of golang code, out of which $2.7k$ lines of code are the implementation of SNARK circuits. This number excludes the adapted Golang TLS standard library. To facilitate parsing of HTTP requests and responses over TLS, as well as to transcribe the TLS handshake and extract relevant intermediate values, we customize the standard TLS package as provided in `go/crypto` [6]. We rely on *gnark* [17] for our implementation of general-purpose SNARKs. Gnark is a library that provides a programming interface to facilitate the development of ZKP applications. It relies on gnark-crypto [18], a library for arithmetic in finite fields and operations over common (pairing-friendly) elliptic curves. Gnark supports implementations of three SNARKs - Groth16, Plonk with the KZG polynomial commitment and Plonk with FRI as a commiment scheme and interactive proof. Our implementation includes circuits for proving SHA-256, HMAC, HMAC with MiMC, AES-128, AES in Galois Counter Mode, and the full TLS key derivation.

We open-source our implementation of ORIGO in a public repository [10]. ORIGO can either be run in a local environment with a mock server, or in a realistic setting with an externally hosted server supporting TLS 1.3. To demonstrate a realistic setting, we integrated ORIGO with the PayPal API [7]. We further depict a practical implementation of our protocol in Appendix A, showing how to obtain attestations in Gitcoin Passport [5].

7 EVALUATION

We evaluate the performance of ORIGO by evaluating *(i)* the overhead of ORIGO over the TLS baseline handshake and request execution, *(ii)* the individual performance of techniques introduced in ORIGO (§ 5.1-§ 5.3) and *(iii)* a realistic, global client-side deployment and *(iv)* a holistic evaluation with regard to related works.

Experimental Setup. Experiments to measure the circuit runtime are conducted on a MacBook Air configured with 24GB of RAM and the Apple M2 chip. In the global deployment, each client is executed on an AWS EC2 c5.4xlarge instance with 32GB of RAM and and Intel Xeon Platinum 8124M CPU with 16 vCPUs. We leverage the insight of Ernstberger *et. al* [30], which states that circuits in gnark are 40% – 50% faster on compute optimized hardware. The proxy runs on an Intel server with 128GB of RAM and an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz. All results are averaged over 20 executions. In our experiments with adjusted bandwidth and throughput, we throttle bandwidth and throughput with `tc`.

Preprocessing. We do not evaluate the pre-processing on the client and proxy side, which are the out of circuit operations as described in Section 5.1. Omitting preprocessing is acceptable, as it is performed out of circuit and only requires a minimal amount of compute and bandwidth. Further, preprocessing is concretely faster than the generation of the Zero Knowledge Proof.

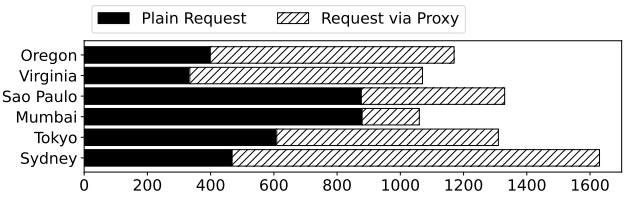

Figure 12: Overhead required for tunneling a TLS session through a proxy. The proxy is fixed in Munich, the client is in 6 different cities. The execution time is in milliseconds. The TLS server is hosted by PayPal [7].

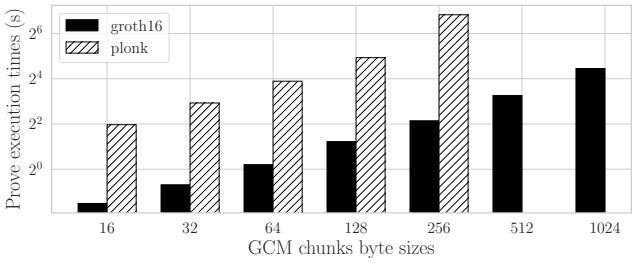

Figure 13: Prover Time for AES in Galois Counter Mode. The circuit is executed with *(i)* a Groth16 backend and *(ii)* a Plonk backend. The x-axis depicts the byte size of the plaintext.

7.1 Overhead over TLS in request execution

Figure 12 shows that the overhead introduced by the proxy setting is minimal with regard to the overall protocol runtime. We measure the latency averaged over 20 invocations with the real-world API provided by PayPal. We deploy the client in 6 distinct locations around the globe, and execute *(i)* a plain request to the API server as well as *(ii)* a handshake and request through the proxy, where the proxy save the relevant handshake transcript as well as the query and response transmitted by the client and server respectively. We find that the overhead introduced by the proxy ranges between 1.2× in Mumbai and 3.5× in Sydney, as opposed to a plain TLS connection the server hosted by PayPal. The overall end-to-end runtime for an HTTP request with ORIGO is 1*s* to 1.6*s*, which is well below application-specific TLS timeouts (10 − 15 seconds).

7.2 Performance of Circuit Implementations

Table 3 in the appendix provides microbenchmarks for the circuits relevant to ORIGO. We measure the size of the circuit in number of constraints, offline and online runtime, and offline and online communication. Note, that offline communication differs depending on the protocol setting and proof system used. For example, for Groth16 there is no separation of a randomized and deterministic setup procedure. Hence, the trusted setup would be executed by an external party, and the prover key and verifier key can be dispersed to the prover and verifier respectively. On the other hand, the randomized setup procedure in Plonk can be executed separately (see e.g. [50] or [8] for a practical example). The SRS can be dispersed, whereas the prover and verifier derive the prover and verifier key locally in a determinstic setup. The complete end-to-end circuit for ORIGO counts 896.3k constraints. We find that the bottleneck in generating the proof is the AES encryption in Galois Counter Mode, which counts 288.1k constraints for encrypting a 32 Byte plaintext and takes 0.61 seconds (cf. Figure 13).

Client Side Evidence Generation. To measure the performance of client side evidence generation, we evaluate the time it takes to *(i)* generate the salt tree and *(ii)* generate the salted merkle tree. In an unoptimized setting the HMAC circuit with the MiMC hash function takes up 52.2k constraints and 0.41s, whereas a comparable SHA-256 based HMAC evaluation takes up 184.9k constraints and 0.75s. Each expansion function requires a single HMAC evaluation for MiMC over BN254, which totals the prover time to ∼ 419s - a significant overhead requiring further optimization.

In case counter blocks are utilized in an optimized evidence generation, only the merkle tree over the encrypted counter blocks needs to be computed in-circuit in addition to the circuits described in Table 3 in the appendix. The generation of an in-circuit Merkle Tree with 1024 leaves of depth 10 takes 2.04s with a MiMC hash, and 0.12s when instantiated recursively with an inner GKR proof and an outer Groth16 proof as desribed by Belling *et. al* [16]. For both, AES encryption remains the bottleneck in overall proof generation. Utilizing lookup arguments to prove AES as described by Orru *et. al* can further lower the cost of AES [51].

7.3 Global Deployment

In Figure 14, we present a comprehensive evaluation of the execution times experienced by clients distributed across the globe when accessing data from a server, using our protocol, via a proxy verifier located in Munich. All clients are running on the same type of AWS instance. We benchmark the performance of our protocol by integrating it with the real-world TLS server hosted by PayPal [7]. The request sent aims to create an order for a product valuated at a specific price. An order in PayPal is an agreement between a buyer and a seller, the integration with ORIGO allows the client to prove that a product has been purchased via PayPal payment for a certain price. This API has a request size of 873 byte and a response size of 1429 byte. We deploy the client in 6 different cities and observe that the online execution time of our protocol ranges from 1.44s to 2.05s. Xie *et. al* [63] conduct a similar experiment by deploying the client in 18 different cities, reporting an online runtime between 0.3s and 9.6s for a similarly sized request / response pair, which indicates that ORIGO enjoys a more stable online execution time, independent of the geographical location of the TLS session verifier. This is primarily caused by the number of online communication rounds, which is 1 for ORIGO, and 31 in the case of Xie *et. al.*

7.4 Comparison with Related Works

We compare the performance of ORIGO to DECO [67] and the concurrent work of Xie *et. al* [63]. Both works rely on a three-party handshake, which replaces the traditional TLS handshake, and do allow for a stronger network adversary as opposed to our protocol. For the measurements of DECO, we rely on the performance evaluation conducted by Xie *et. al* [63], which instantiate DECO with the state-of-the-art protocol for malicious 2PC, Ferret [64]. Note, that the evaluation of DECO only includes the time needed in malicious

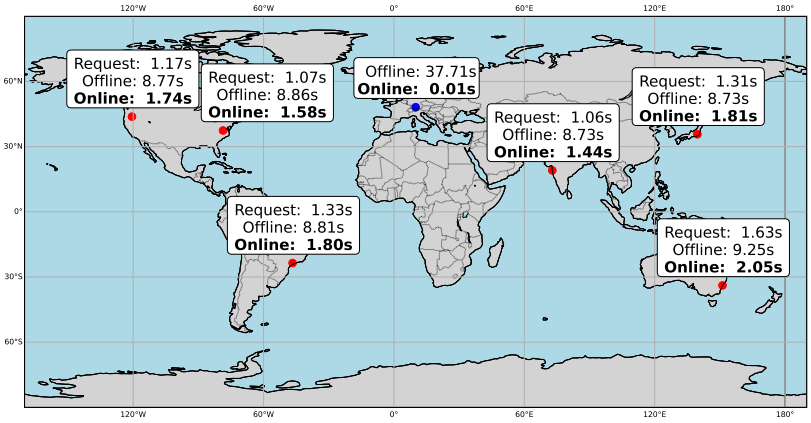

Figure 14: Execution time for globally dispersed clients requesting data from a PayPal server through a proxy verifier situated in Munich. For the proxy, the "Offline" time includes the setup of pk & vk, and compiling the constraint system. For clients, the "Offline" time is equivalent to compiling the constraint system. The "Online" execution time is the time to create the proofs as described in Section 5, verifying them on the proxy side, and receiving a response.

Table 2: Comparing DECO, XYWY23, and this work under WAN conditions. For communication cost, values in brackets signify the communication cost for prover/verifier keys respectively. For runtime, we report online operations; the values in brackets are offline operations for the SNARK setup and compilation of the circuit. The proof system used in our work is Groth16.

	Communication cost			WAN (100Mbps, RTT=50ms)		
	256 B	1 KB	2 KB	256 B	1 KB	2 KB
DECO [67]	206 MB	345 MB	476 MB	24 s	36.3 s	51.6 s
XYWY23 [63]	15.2 MB	22.9 MB	33.3 MB	3.19 s	3.96 s	4.9 s
This Work	**2.4 kB** (111.9 MB / 10.9 kB)	**2.4 kB** (111.9 MB / 10.9 kB)	**2.4 kB** (111.9 MB / 10.9 kB)	**1.50 s** (28.6 s / 9.2 s)	**1.53 s** (28.6 s / 9.2 s)	**1.51 s** (28.6 s / 9.2 s)

2PC, which includes computing the TLS session keys and 4 pairs of AES-GCM ciphertext/tags [63]. All performance numbers are measured using the same type of AWS instance (AWS c5.4xlarge). We summarize the results of our comparison in Table 2.

As expected, we observe that our protocol reports constant communication cost and constant execution time, independent of the size of the request and response. For communication cost, we can observe that the required online communication in ORIGO is smaller than in both DECO and the work of Xie *et. al* [63]. For a query size of 587 bytes and response size of 894 bytes, Xie *et. al* report a communication cost of 17.6 MB offline and 0.9 MB online, whereas ORIGO requires a constant 2.4 kB, which is 375× less. For a request and response of 256 Byte, our solution completes in 1.50 seconds, which is significantly faster than DECO's 24 seconds. This presents an approximate 16× speedup. Note, that the reported online execution time highly depends on the size of the statement to be proven. In our example, the statement fits into two chunks of 128 bits. If the statement to be proven requires in-circuit encryption of multiple chunks, the online time of ORIGO scales as described in Section 7.2.

8 DISCUSSION AND FUTURE WORK

This paper introduces ORIGO, a TLS oracle with constant communication built on general purpose ZKPs. The main challenge is to minimize the in-circuit operations to achieve practical prover runtimes (§ 5.1) to ensure mitigation of an attack on integrity (§ 4). Another challenge was achieving flexibility for clients, such that they can create arbitrary proofs post-session with minimal overhead (§ 5.3). We show that the total online communication of ORIGO is 2.4kB. We evaluate our protocol in a practical setting with globally dispersed clients, showing that the online runtime for proof generation and presentation ranges between 1.44 seconds and 2.05 seconds. The performance benefit is primarily obtained by the proxy setting, and the assumption that MITM attacks are infeasible, which allows us to use a SNARK in a public coin protocol. This is interesting, as it allows outsourcing the prover to an untrusted server without forfeiting confidentiality [35].

There are three limitations of ORIGO that demand for further exploration. First, assuming that MiTM is impossible demands for additional mitigation on other OSI layers. Certain applications, like Gitcoin Passport, may deem this tradeoff sufficient for their

application, as practically instantiating an attack is non-trivial. For the client to perform the attack, it would need to manipulate the DNS resolution at the proxy, or intercept the traffic between the proxy and the actual server. This could be attempted through DNS hijacking or spoofing. Doing so would require compromising the DNS resolver used by the proxy. By using DNSSEC, the proxy can ensure that the domain name resolution is secure and authentic. To ensure that the client doesn't supply a malicious domain name in its request, the proxy can pre-specify a set of whitelisted domain names. We further discuss the feasibility of an MITM attack for traffic injection in Appendix D. Second, the size of the SRS is large ($\sim 100 - 300$MB for Groth16/Plonk) and requires additional offline communication. Albeit constant, it is unclear how clients receive an SRS. If each proxy verifier creates its own SRS, clients need to download large files before executing a session. Usage of established SRS, with a sufficient amount of public randomness in a public MPC setup ceremony [8], may be a suitable alternative to lessen this issue. Third, AES encryption remains the bottleneck in circuit compilation and proof generation. Recent work shows how to minimize AES constraints with lookup arguments [51]. We believe instantiating their construction in-circuit to be clear items for future work.

REFERENCES

[1] 2007. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. https://csrc.nist.gov/publications/detail/sp/800-38d/final.

[2] 2014. TLS Notary - Initial Version. https://old.tlsnotary.org/how_it_works.

[3] 2014. tlsnotary - cryptographic proof of fiat transfer for p2p exchanges. https://bitcointalk.org/index.php?topic=173220.0.

[4] 2023. Ceramic Network. https://ceramic.network/.

[5] 2023. Gitcoin Passport. https://passport.gitcoin.co/.

[6] 2023. Golang Crypto Package. https://pkg.go.dev/crypto.

[7] 2023. PayPal API specification. https://developer.paypal.com/api/rest/.

[8] 2023. PROTO-DANKSHARDING AND THE CEREMONY. https://ceremony.ethereum.org/.

[9] 2023. ThousandEyes - Best BGP Route Network Monitoring Solution. https://www.thousandeyes.com/solutions/bgp-and-route-monitoring.

[10] 2023. TLS Oracle Demo. https://github.com/opex-research/tls-oracle-demo.

[11] Behzad Abdolmaleki, Noemi Glaeser, Sebastian Ramacher, and Daniel Slamanig. 2023. Universally Composable NIZKs: Circuit-Succinct, Non-Malleable and CRS-Updatable. *Cryptology ePrint Archive* (2023).

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

[13] Oussama Amine, Karim Baghery, Zaira Pindado, and Carla Ràfols. 2023. Simulation extractable versions of Groth's zk-SNARK revisited. *International Journal of Information Security* (2023), 1–15.

[14] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. 2021. Another look at extraction and randomization of groth's zk-SNARK. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I 25*. Springer, 457–475.

[15] Mihir Bellare and Oded Goldreich. 1992. On defining proofs of knowledge. In *Annual International Cryptology Conference*. Springer, 390–420.

[16] Alexandre Belling, Azam Soleimanian, and Olivier Bégassat. 2022. Recursion over Public-Coin Interactive Proof Systems; Faster Hash Verification. *Cryptology ePrint Archive* (2022).

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

[18] Gautam Botrel, Thomas Piellard, Youssef El Housni, Arya Tabaie, Gus Gutoski, and Ivo Kubjas. 2023. *ConsenSys/gnark-crypto: v0.11.2.* https://doi.org/10.5281/zenodo.5815453

[19] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.

[20] Ran Canetti and Marc Fischlin. 2001. Universally composable commitments. In *Advances in Cryptology—CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings 21*. Springer, 19–40.

[21] Srdjan Capkun, Ercan Ozturk, Gene Tsudik, and Karl Wüst. 2021. ROSEN: RObust and SElective Non-repudiation (for TLS). In *Proceedings of the 2021 on Cloud Computing Security Workshop*. 97–109.

[22] Sofía Celi, Alex Davidson, Hamed Haddadi, Gonçalo Pestana, and Joe Rowell. 2023. Distefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more. *Cryptology ePrint Archive* (2023).

[23] Kwan Yin Chan, Handong Cui, and Tsz Hon Yuen. 2023. DIDO: Data Provenance from Restricted TLS 1.3 Websites. *Cryptology ePrint Archive* (2023).

[24] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. 2020. Fractal: Post-quantum and transparent recursive proofs from holography. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*. Springer, 769–793.

[25] World Wide Web Consortium et al. 2019. Verifiable credentials data model 1.0: Expressing verifiable information on the web. *https://www. w3. org/TR/vc-data-model/?# core-data-model* (2019).

[26] Tim Dierks and Eric Rescorla. 2008. The transport layer security (TLS) protocol version 1.2. (2008).

[27] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. 2018. Fast message franking: From invisible salamanders to encryptment. In *Annual International Cryptology Conference*. Springer, 155–186.

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

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

[30] Jens Ernstberger, Stefanos Chaliasos, George Kadianakis, Sebastian Steinhorst, Philipp Jovanovic, Arthur Gervais, Benjamin Livshits, and Michele Orrù. 2023. zk-Bench: A Toolset for Comparative Evaluation and Performance Benchmarking of SNARKs. *Cryptology ePrint Archive* (2023).

[31] Jens Ernstberger, Jan Lauinger, Fatima Elsheimy, Liyi Zhou, Sebastian Steinhorst, Ran Canetti, Andrew Miller, Arthur Gervais, and Dawn Song. 2023. SoK: Data Sovereignty. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 122–143. https://doi.org/10.1109/EuroSP57164.2023.00017

[32] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. 2016. Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 452–469.

[33] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive* (2019).

[34] Chaya Ganesh, Yashvanth Kondi, Claudio Orlandi, Mahak Pancholi, Akira Takahashi, and Daniel Tschudi. 2023. Witness-succinct universally-composable snarks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 315–346.

[35] Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. 2023. zkSaaS: Zero-Knowledge SNARKs as a Service. *Cryptology ePrint Archive* (2023).

[36] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. 2015. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)* 62, 4 (2015), 1–64.

[37] Jens Groth. 2006. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In *Advances in Cryptology–ASIACRYPT 2006: 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006. Proceedings 12*. Springer, 444–459.

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

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

[40] Rawane Issa, Nicolas Alhaddad, and Mayank Varia. 2022. Hecate: Abuse reporting in secure messengers with sealed sender. In *31st USENIX Security Symposium (USENIX Security 22)*. 2335–2352.

[41] Peng Jiang, Baoqi Qiu, and Liehuang Zhu. 2022. Report When Malicious: Deniable and Accountable Searchable Message-Moderation System. *IEEE Transactions on Information Forensics and Security* 17 (2022), 1597–1609.

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

[43] Eleftherios Kokoris Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Svetolik Jovanovic, Ewa Syta, and Bryan Alexander Ford. 2021. Calypso: Private data management for decentralized ledgers. *Proceedings of the VLDB Endowment* 14, CONF (2021), 586–599.

[44] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, Abhi Shelat, and Elaine Shi. 2015. Coco: a framework

for building composable zero-knowledge proofs. *Cryptology ePrint Archive, Report 2015/1093* (2015).
[45] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. 1997. HMAC: Keyed-hashing for message authentication.
[46] Jan Lauinger, Jens Ernstberger, Andreas Finkenzeller, and Sebastian Steinhorst. 2023. Janus: Fast Privacy-Preserving Data Provenance For TLS 1.3. *Cryptology ePrint Archive* (2023).
[47] Jiqiang Lu and Jongsung Kim. 2008. Attacking 44 rounds of the SHACAL-2 block cipher using related-key rectangle cryptanalysis. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 91, 9 (2008), 2588–2596.
[48] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Ruzica Piskac, and Mariana Raykova. 2023. Privacy-Preserving Regular Expression Matching using Nondeterministic Finite Automata. *Cryptology ePrint Archive* (2023).
[49] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. 2021. Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1348–1366.
[50] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. 2022. Powers-of-tau to the people: Decentralizing setup ceremonies. *Cryptology ePrint Archive* (2022).
[51] Michele Orrù, George Kadianakis, Mary Maller, and Greg Zaverucha. 2024. Beyond the circuit: How to Minimize Foreign Arithmetic in ZKP Circuits. *Cryptology ePrint Archive* (2024).
[52] Deevashwer Rathee, Guru Vamsi Policharla, Tiancheng Xie, Ryan Cottone, and Dawn Song. 2022. ZEBRA: Anonymous Credentials with Practical On-chain Verification and Applications to KYC in DeFi. Cryptology ePrint Archive, Paper 2022/1286. https://eprint.iacr.org/2022/1286 https://eprint.iacr.org/2022/1286.
[53] Michael Raymond, Gillian Evers, Jan Ponti, Diya Krishnan, and Xiang Fu. 2023. Efficient Zero Knowledge for Regular Language. *Cryptology ePrint Archive* (2023).
[54] Eric Rescorla. 2018. *The transport layer security (TLS) protocol version 1.3.* Technical Report.
[55] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, and Srdjan Capkun. 2017. TLS-N: Non-repudiation over TLS enabling-ubiquitous content signing for disintermediation. *Cryptology ePrint Archive* (2017).
[56] Martin Schläffer. 2011. *Cryptanalysis of AES-based hash functions.* na.
[57] Chris McMahon Stone, Tom Chothia, and Flavio D Garcia. 2017. Spinner: Semi-automatic detection of pinning without hostname verification. In *Proceedings of the 33rd annual computer security applications conference*. 176–188.
[58] Nirvan Tyagi, Paul Grubbs, Julia Len, Ian Miers, and Thomas Ristenpart. 2019. Asymmetric message franking: Content moderation for metadata-private end-to-end encryption. In *Annual International Cryptology Conference*. Springer, 222–250.
[59] Yinxin Wan, Kuai Xu, Guoliang Xue, and Feng Wang. 2020. Iotargos: A multi-layer security monitoring system for internet-of-things in smart homes. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 874–883.
[60] Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and Abhi Shelat. 2019. Blind certificate authorities. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1015–1032.
[61] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1074–1091.
[62] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. 2022. AntMan: interactive zero-knowledge proofs with sublinear communication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2901–2914.
[63] Xiang Xie, Kang Yang, Xiao Wang, and Yu Yu. 2023. Lightweight Authentication of Web Data via Garble-Then-Prove. *Cryptology ePrint Archive* (2023).
[64] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. 2020. Ferret: Fast extension for correlated OT with small communication. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1607–1626.
[65] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. 2023. Zombie: Middleboxes that Don't Snoop. *Cryptology ePrint Archive* (2023).
[66] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*. 270–282.
[67] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. 2020. DECO: Liberating web data using decentralized oracles for TLS. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1919–1938.
[68] Jian Zhang, Jennifer Rexford, and Joan Feigenbaum. 2005. Learning-based anomaly detection in BGP updates. In *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*. 219–220.

A APPLICATION TO GITCOIN PASSPORT

To demonstrate the practical use of our construction, we integrated ORIGO with Gitcoin Passport [5]. With Gitcoin Passport, users collect *stamps* from authenticators such as Facebook, Twitter or Github. By collecting stamps, a user can prove *sybil-resistance*. The more stamps a user collects, the higher the trust score, which indicates how difficult it is to forge an identity. To issue a stamp, a user has to authorize the Gitcoin Passport application to write to its personal Ceramic datastream[1] ("Passport"). The user connects to the Gitcoin Passport app, which communicates with the Gitcoin Identity and Access Management (IAM) server, to verify that an account with one of the previously mentioned platforms exists. The Gitcoin passport application, in collaboration with the Gitcoin IAM server, acts as centralized issuer of a verifiable credential ("stamps") as specified by the standardization of the W3C [25]. Once a stamp is issued, anyone can verify its validity. Note that presently, all stamps issued through Gitcoin Passport rely on publicly available data.

With ORIGO, users can create stamps from their private data, without relying on an centralized entity which is trusted for confidentiality. We integrated ORIGO with Gitcoin Passport to create a stamp from a successful PayPal payment through the PayPal API. To initiate the session, the client signs up with the PayPal API to generate an access token. The client executes Π_{ORIGO} with the proxy as a verifier, the PayPal API acts as the server and Gitcoin/Ceramic acts as an external third party. After obtaining a response to the HTTPS GET request, the client proves that the response submitted via TLS contains information about a payment, that is greater than a certain threshold, to the proxy verifier. Upon verification of message integrity and the correctness of values, the proxy verifier creates a signed verifiable credential[2] and sends it to the client. We leverage an adapted version of the writer SDK of Gitcoin Passport[3], such that the client can create and write to its personal Ceramic Tile Stream [4]. The verifiable credential is then included as a stamp under the previously created DID. As such, our application allows for stamps from private data, where the user does not give up control over its "passport" to another entity. By issuing verifiable credentials from private data, the proxy verifier can act as a complementary issuer to the Gitcoin Passport application, without being trusted for confidentiality.

B PRIVACY RIGHTS & REGULATIONS

In our practical implementation, each session in ORIGO is governed by a *policy*. The purpose of a policy for data feeds for confidential data is twofold — to provide a set of constraints that enables the verifier to validate the correctness of statements of the client, and to provide an interface that ensures the validity of data sources providing qualitatively equivalent data (e.g., age verification through either the DMV or governmental API endpoints). In our current implementation, the policy specifies applicable data sources, response formats and value constraints to be proven by the client.

Further, a policy can also implicitly or explicitly guarantee data accuracy. To address liability concerns, the policy could include

[1] A data storage network for Web 3.0 applications which identifies data streams by a *Decentralized Identifier (DID)*.
[2] We integrate a JavaScript SDK for W3C Verifiable Credentials https://github.com/digitalbazaar/vc-js
[3] https://github.com/gitcoinco/passport-sdk/tree/main/packages/writer

Table 3: Benchmarks for SNARK circuits in ORIGO. We assume the context spans 32 Byte, i.e. two TLS record blocks need to be encrypted in-circuit. We separate offline and online runtime and communication. CCS is the *compiled constraint system*, SRS is the *reference string* established in the randomized setup of the SNARK, PK and VK are the *prover key* and *verifier key* which are derived in a determinstic setup, PW is the *public witness*, and proof π.

		Execution Offline		Execution Online		Communication Offline			Communication Online	
ZKP Circuit	# Constraints (x10^3)	Compile (s)	Setup (s)	Prove (s)	Verify (s)	CCS, SRS (MB)	PK (MB)	VK (KB)	π (KB)	PW (B)
Groth16 backend										
KDC	322.4	0.9	11.12	0.48	0.001	35.3	47.8	4.93	0.128	128
Tag	285.8	1.28	9.17	0.6	0.001	106.6	40.1	2.37	0.128	44
Record	288.1	1.31	9.24	0.61	0.001	105.1	40.4	2.02	0.128	72
Plonk backend										
KDC	573.0	7.64	6.08	7.38	0.002	52.7, 33.5	327.1	0.54	0.552	128
Tag	551.9	7.65	6.14	7.46	0.002	61.5, 33.5	327.1	0.54	0.552	44
Record	556.5	7.83	6.16	7.67	0.002	61.9, 33.5	327.1	0.54	0.552	72

terms that specify the extent of the data provider's liability for data accuracy. Moreover, including a hash of a Terms of Service (ToS) or Disclaimer within the policy can serve as an immutable reference to the terms under which the data is provided and attested.

Thus, ORIGO can enforce terms under which data is attested, and deny attestations that violate these terms. We expect that a set of policies will provide clients with a choice of data providers and endpoints, from which attested data can be obtained – similar to how passport stamps can be obtained in Gitcoin Passport [5].

C FEASIBILITY OF THE ATTACK ON TLS 1.3

The attack presented in Section 4 challenges the integrity and uniqueness aspects of AES-GCM, aiming to compromise its security by finding collisions in encryption and authentication. The attack applies in a setting, where the adversary aims to convince an entity about an underlying plaintext, whereas the entity verifying the statement already holds the ciphertext encrypted with AES-GCM. The attack leverages that in TLS 1.3, the IV is *not* explicitly included in the transmitted messages, but rather derived from the handshake transcript. We begin to outline the trivial attack on an arbitrary plaintext, and successively discuss the feasibility of the attack given an adversarial chosen, structured plaintext.

First, the adversary $\mathcal{A}_{GCM}$ a key K'. Successively, $\mathcal{A}_{GCM}$ computes the evaluation of the binary polynomial at H, by computing

$$X \leftarrow \sum_{i=1}^{m} C_i H^{m-i}$$

where $H \leftarrow E_K\{0^{128}\}$ is the encryption of the zero vector. Here, $\mathcal{A}_{GCM}$ is unable to choose X, as each C_i is fixed (i.e. known by the verifying entity). Now, $\mathcal{A}_{GCM}$ chooses IV' such that $T = T'$, i.e. the newly computed tag is equivalent to the one known by the verifying entity for a differing, adversarial chosen initialization vector. T' is computed as $T' \leftarrow E_{K'}\{IV'\} \oplus X$. Now that an adversarial tag for a differing IV is computed, it remains to show that the adversary can derive a colluding ciphertext with a differing plaintext, given IV' and K'. To do so, the adversary computes $R' \leftarrow E_{K'}\{IV'||ctr\}$ and chooses an arbitrary plaintext P such that $C \leftarrow R' \oplus P'$ is satisfied. Given an arbitrary plaintext, this attack is hence trivial.

For a chosen plaintext, $\mathcal{A}_{GCM}$ needs to find a second preimage attack where they generate a different set of inputs (K', IV', P') that produce the same encrypted output and authentication tag as the original (K, IV, P). While an adversary can freely choose a new key and IV, ensuring these selections result in the same tag and ciphertext for a different plaintext requires reversing the GHASH and AES. For $\mathcal{A}_{GCM}$ to revert AES-GCM, the expected number of attempts is in the order of 2^{128}, which is the size of the key space. Previous work on cryptanalytic techniques demonstrate a rebound attack that can, under certain conditions, significantly lower the complexity of finding a set of inputs that satisfy a given cryptographic condition than what a brute-force approach would suggest [56]. For AES-based functions, this attack lowers the complexity to about 2^{48} for constructing valid pairs that follow a specific differential trail. We consider a concrete analysis with cryptanalytic techniques out of scope and interesting for future research.

D ON THE POSSIBILITY OF MITM ATTACKS

Assuming that MITM by a malicious client is not possible demands for additional mitigation on another OSI layer (without any server-side changes). For example, a proxy can employ IP Address Monitoring [59] and Secure BGP Practices on the network layer [9, 68] or Certificate Pinning on the transport layer to ensure correct certificates [57]. Further, external verification of the server's public key, separate to the proxy session, can ascertain additional security.

Similar assumptions, which assume additional care to account for traffic injection through e.g. BGP attacks, have been acknowledged and embraced in similar settings that require a proxy [60, 67]. The reason for this is that BGP attacks are *(i)* are difficult to mount in practice, and *(ii)* various detection techniques have been proposed to detect BGP attacks [9, 68]. To additionally enhance the security of the approach introduced in this work, one may rely on an N-out-of-N multisignature with globally dispersed proxies, or a similar approach that relies on N-version programming, following the initial specification outlined in this work. However, we acknowledge

$\mathcal{F}_{DP}(\lambda, \mathcal{R}, \mathcal{S}, \mathcal{V}, \mathcal{C})$

\\ Initialize
Init: Set Storage $:= \emptyset$ and initialize the corruption state as uncorrupted.
\\ Client sends request
On receive (sid, "request", $\theta_{\mathcal{S}}$) from $\mathcal{C}$:
  set Q = Request($\theta_{\mathcal{S}}$)
  notify $\mathcal{A}$ of (sid, "request", $|Q|$)
  set Storage[sid] $:= (Q, \_)$
\\ Server send response
On receive (sid, "response", Q, R) from $\mathcal{S}$:
  set Q' from Storage[sid]
  check that $Q' = Q$, else return $\perp$
  notify $\mathcal{A}$ of (sid, "response", $|R|$)
  update Storage[sid] $:= (Q, R)$
\\ Forward any data
On receive (sid, "forward", $\mathcal{P}$) from $\mathcal{V}$:
  if $\mathcal{P} = \mathcal{S}$ set out = Q from Storage[sid]
  if $\mathcal{P} = \mathcal{C}$ set out = R from Storage[sid], else return $\perp$
  notify $\mathcal{A}$ of (sid, "forward", |out|, $\mathcal{P}$), block until $\mathcal{A}$ replies
  send (sid, out) to $\mathcal{P}$
\\ Client proves provenance (and statement, optional)
On receive (sid, "prove", R) from $\mathcal{C}$:
  set R' from Storage[sid]
  check that $R' = R$, else return $\perp$
  send (sid, $\mathcal{R}(R)$) to $\mathcal{V}$
On receive ("corrupted") from $\mathcal{V}$
  set corruption state to corrupted and return an element
  randomly sampled from $\{0,1\}^{\lambda}$ when invoked with "forward"

Figure 15: The ideal functionality for data provenance via TLS with a proxy verifier $\mathcal{V}$.

that additional MiTM mitigation mechanisms introduce an additional overhead to our protocol when compared with approaches that do not face this issue [22, 46, 63, 67], which our evaluation does not account for. We leave further exploration of extending the security guarantees in the proxy mode as future work.

E FORMAL SECURITY ANALYSIS

In this section, we analyze and prove the security of Π_{ORIGO} in the UC paradigm by Cannetti *et. al* [19]. In short, the UC model employs a real-ideal world paradigm, where security follows from indistinguishability between an execution of the real protocol in the real world and execution of an ideal version in the ideal world. The reasoning is that the ideal functionality specifies the ideal security properties and fulfills them by definition. If the real protocol execution is indistinguishable from the execution with the ideal world, it follows that it yields exactly the same security properties.

Formally, a protocol Π UC-realizes and ideal functionality $\mathcal{F}$ if Π emulates $I_{\mathcal{F}}$, the ideal protocol of $\mathcal{F}$. A protocol Π emulates protocol $I_{\mathcal{F}}$ if for any polynomial time (PT) adversary $\mathcal{A}$, there exists a PT adversary Sim such that for all PT environments $\mathcal{Z}$ that only output one bit the following executions are indistinguishable:

$$\forall \mathcal{Z}; EXEC_{PROT,\mathcal{A},\mathcal{Z}} \approx EXEC_{I_{\mathcal{F}},\mathtt{Sim},\mathcal{Z}} \tag{2}$$

E.1 Ideal Functionality $\mathcal{F}_{DP}$

We model the security of proving data provenance as an ideal functionality $\mathcal{F}_{DP}$ in Figure 15. Each participant in the ideal functionality is identified by a unique identifer, denoted as $\mathcal{S}$, $\mathcal{V}$, $\mathcal{C}$ and $\mathcal{P}$ respectively. Further, $\mathcal{F}_{DP}$ is parametrized with a relation $\mathcal{R}$ to be satisfied by the data provided by the server $\mathcal{S}$. The unique identification of participants models the requirement of uniquely identifiable communication partners, which is a requirement to account for the possibility of adversarial MITM clients. Note that in the UC model, identifiers assumed to be provided via an underlying lower-level protocol that also guarantees authenticated communication [20]. The network adversary is formalized through the delayed output terminology [19]. At a high level, the client $\mathcal{C}$ first calls the "request" subroutine of $\mathcal{F}_{DP}$ by providing a authentication secret $\theta_{\mathcal{S}}$. The functionality internally stores the query Q for the session identifier sid. Note that only once the proxy verifier $\mathcal{V}$ calls "forward", the request is forwarded to the server $\mathcal{S}$. However, the verifier $\mathcal{V}$ never learns the content request. The adversary solely learns the size of the request Q. Once the server $\mathcal{S}$ receives the response, it checks whether $\mathcal{V}$ actually forwarded the request sent by $\mathcal{C}$. Similarly, once the client receives the response R, it checks whether $\mathcal{V}$ actually forwarded the response sent by $\mathcal{S}$. Intuitively, $\mathcal{F}_{DP}$ describes implies the following security properties, which are similarly described in related works [63, 67]: **Client Integrity.** A malicious client cannot convince a verifier of an untrue statement about the TLS session. **Verifier Integrity.** A malicious verifier cannot prevent a client from receiving the correct message from the server. **Privacy.** A malicious proxy verifier cannot obtain any knowledge beyond the size of the message that it forwards.

E.2 Security Proof

In the following, we show the validity of Theorem 1. For clarity, we highlight that Theorem 1 presumes EUF-CMA security of ECDSA for verification of certificates, and security of the SHACAL-2 compression function f underlying the SHA-256 hash function, as only the input to the first compression function of the Merkle-Damgard structure is proven in our circuit. To prove that Π_{ORIGO} UC-realizes $\mathcal{F}_{DP}$, we specify a simulator Sim, such that no environment $\mathcal{Z}$ can distinguish an interaction between Π_{ORIGO} and $\mathcal{A}$ from an interaction with $\mathcal{F}_{DP}$ and Sim.

Construction of Sim. We construct the simulator Sim by distinguishing the two main cases for adversarial entities in Π_{ORIGO} – an adversarial $\mathcal{C}$ and an adversarial $\mathcal{V}$. We assume the ideal functionality as presented in Figure 15. As the server $\mathcal{S}$ is always presumed to be honest, we distinguish between the simulator for *(i)* an adversarial $\mathcal{C}$ and *(ii)* an adversarial $\mathcal{V}$ below. Sim runs $\mathcal{A}_{\mathcal{C}}$ (and for *2)* $\mathcal{A}_{\mathcal{V}}$), as well as $\mathcal{F}_{ZK}$, internally. Sim forwards any input from $\mathcal{Z}$ and tracks messages sent from and to $\mathcal{A}_{\mathcal{C}}$ / $\mathcal{A}_{\mathcal{V}}$. The simulator Sim simulates the compression function f as a random oracle by sending uniformly random strings from $\{0,1\}^{256}$ when queried by the adversary.

1) Sim for malicious $\mathcal{C}$. For a malicious client $\mathcal{C}$ we intend to prove *client integrity*.
(1) When $\mathcal{A}$ sends (sid, "request", Q) to $\mathcal{S}$, Sim forwards (sid, "forward", $\mathcal{S}$) to $\mathcal{F}_{DP}$. When server sends (sid, "response", R) to $\mathcal{C}$, Sim forwards (sid, "forward", $\mathcal{C}$) to $\mathcal{F}_{DP}$ and (sid, out) to $\mathcal{A}$. Sim obtains $(\hat{Q}, \hat{R})$, where $\hat{Q}$ is the encrypted request and $\hat{R}$ is the encrypted response.
(2) When receiving (sid, "prove", C, x, w) from $\mathcal{A}$, Sim checks that $k := \text{KDC}(\Gamma)$ for with the TLS transcript and $\hat{R} := \text{Enc}_k(R)$ with AES-GCM.

(3) If the previous checks pass, Sim forwards (sid, "prove", R) to $\mathcal{F}_{DP}$, which successively outputs to $\mathcal{V}$. Sim outputs whatever $\mathcal{A}$ outputs.

Indistinguishability. We now argue that the execution with Sim and $\mathcal{F}_{DP}$ is indistinguishable from the real-world execution with $\mathcal{A}$ and Π_{ORIGO} through a set of hybrids. We start off with hybrid $H1$, which represents the real-world execution of the protocol. $H2$ lets Sim emulate $\mathcal{F}_{ZK}$. $H3$ lets Sim simulates $\mathcal{V}$. $H4$ adds the integrity checks from step *3)*.

Hybrid $H1$**.** Is the real-world execution of Π_{ORIGO}.

Hybrid $H2$**.** proceeds as in $H1$, but in addition the simulator Sim emulates $\mathcal{F}_{ZK}$. All communication between $\mathcal{A}_C$ and $\mathcal{S}$ or $\mathcal{V}$ is sent through Sim, which forwards it as in the real execution. It is trivial to see that $H1$ and $H2$ are indistinguishable as the simulation of $\mathcal{F}_{ZK}$ is perfect.

Hybrid $H3$**.** $H3$ is the same as $H2$, but Sim internally simulates $\mathcal{V}$. Sim only forwards the client request and server response if it was recorded. $H3$ is indistinguishable from $H2$, as the distribution in the ideal and real world are the same, i.e. both $\mathcal{V}$ and Sim would abort under the same circumstances when C presents an invalid proof.

Hybrid $H4$**.** $H4$ is the same as $H3$, with the change that the additional check for record existence is performed by Sim. To show that $H3$ is indistinguishable from $H4$, we show that both $\mathcal{V}$ and Sim would abort in the same cases. In the real world, the misbehavior of the client would lead to an abort in two cases — the client deviates during the handshake with the server or in the proof presentation. We argue that the same two would lead to an abort in $H4$ as follows. First, the client may deviate during handshake. This will be detected by Sim and will result in exactly the same behavior as in the real world. Second, the client may provide a proof on data that is not associated with the record transmitted. This would lead similarly to an abort in the real and ideal world as Sim internally checks whether the sent queries and records match the session id.

It remains to argue that $H4$ is indistinguishable from the ideal execution. As $H4$ only accepts a valid statement if the client is in posession of $\theta_{\mathcal{S}}$ and obtained the records from $\mathcal{S}$, $H4$ and the ideal world are indistinguishable.

2) Sim for malicious $\mathcal{V}$. For a malicious $\mathcal{V}$, we intend to show verifier integrity, such that a malicious $\mathcal{V}$ cannot prevent a client from receiving correct messages from the server. Recall that Sim runs $\mathcal{A}_{\mathcal{V}}$ internally and forwards all traffic between $\mathcal{Z}$ and $\mathcal{A}_{\mathcal{V}}$.

(1) Upon reception of any message sent by $\mathcal{A}_{\mathcal{V}}$, Sim forwards it to $\mathcal{F}_{DP}$. During initialization, Sim obtains the encrypted handshake transcript and forwards it to $\mathcal{A}_{\mathcal{V}}$. The client C decrypts the handshake and verifies $\delta_{\mathcal{S}}$.

(2) After C calls $\mathcal{F}_{DP}$ with "request" or $\mathcal{S}$ calls $\mathcal{F}_{DP}$ with "response", Sim obtains the leaked output sizes $|Q|$ and $|R|$.

(3) Upon request of $\mathcal{A}_{\mathcal{V}}$, Sim runs the handshake with $\mathcal{S}$ by simulating C, obtaining the handshake secrets and traffic encryption secrets negotiated with $\mathcal{S}$.

(4) Sim sends a random request Q to $\mathcal{A}_{\mathcal{V}}$ and forwards to $\mathcal{F}_{DP}$ whatever $\mathcal{A}_{\mathcal{V}}$ responds.

(5) Sim asserts that $\delta_{\mathcal{S}}$ verifies with $pk_{\mathcal{S}}$.

(6) Sim prepares the alternative handshake transcript by pre-computing internal hash functions and forwarding them to $\mathcal{A}_{\mathcal{V}}$. Sim discloses the handshake keys to $\mathcal{A}_{\mathcal{V}}$.

(7) In the proof presentation phase, Sim sends (sid, $\mathcal{R}(R)$) to $\mathcal{A}_{\mathcal{V}}$ and responds whatever $\mathcal{A}_{\mathcal{V}}$ responds.

Indistinguishability. We argue that the ideal execution with Sim and $\mathcal{F}_{DP}$ is indistinguishable from the real-world execution with $\mathcal{A}$ and Π_{ORIGO} through a set of hybrids. $H1$ represents the execution of the real-world protocol. $H2$ lets Sim additionally emulate $\mathcal{F}_{ZK}$. $H3$ additionally lets Sim simulate C.

Hybrid $H1$**.** is the real-world execution of Π_{ORIGO}.

Hybrid $H2$**.** $H2$ is the same as $H1$, but Sim internally simulates $\mathcal{F}_{ZK}$. Further, Sim calls $\mathcal{F}_{DP}$ with "request" to receive $|Q|$ and $|R|$. Since the simulation of the ideal functionality is perfect, $H2$ and $H1$ are indistinguishable.

Hybrid $H3$**.** $H3$ is the same as $H2$, but $H3$ additionally internally simulates C. Sim engages in the handshake with $\mathcal{S}$. Sim emulates a handshake transcript encrypted with random keys and chooses Q, R at random and provides them to $\mathcal{A}_{\mathcal{V}}$. Finally, Sim directly sends (sid, "prove", data, $\mathcal{S}$) to $\mathcal{A}_{\mathcal{V}}$ alongside the pre-computed adapted handshake transcript and the handshake encrypting keys.

It remains to argue that $H3$ is the same as the ideal world. We argue that this is the case because: When providing the alternative transcript, $\mathcal{A}_{\mathcal{V}}$ only learns a hash function in the real world, as f is a random oracle. Further, as EUF-CMA security holds, signed certificates cannot be forged. Due to key independence, $\mathcal{A}_{\mathcal{V}}$ does not learn anything about the record layer from the handshake layer. Records and Queries appear equally random. In the record layer, Q and R are encrypted and of the same size as in the real world, such that they're indistinguishable Finally, $\mathcal{A}_{\mathcal{V}}$ receives the same output as in the real-world execution.

Instantiating $\mathcal{F}_{ZK}$. Universally Composable security of SNARKs has been deeply investigated in previous works [11, 13, 14, 34, 44]. The core problem with ensuring universal composability for SNARKs is that they commonly rely on the Fiat-Shamir transform for non-interactivity and therefore demand a non-black-box extractor. For Groth16, it has been shown that it only satisifies weak simulation extractability [14], a weaker notion than universal composability. Achieving universal composability for e.g. Groth16 demands for custom designs that achieve a stronger notion of simulation extractability with black-box simulation[37]. Recent work shows how to obtain strong simulation extractability for Groth16 with minimal overhead [13]. Further, Abdolmaleki *et. al* [11] provide a generic UC compiler for NIZKs with updatable CRS, such as Plonk, that introduces a $\approx 1.2\times$ overhead in concrete instantiation. Note, that the implementation of Groth16 and Plonk in gnark is not adapted, and hence they do not satisfy $\mathcal{F}_{ZK}$. We leave an instantiation with universally composable SNARKs to future work.