TLS-N: Non-repudiation over TLS
Enabling Ubiquitous Content Signing for Disintermediation

Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, Srdjan Čapkun
Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

An internet user wanting to share observed content is typically restricted to primitive techniques such as screenshots, web caches or share button-like solutions. These acclaimed proofs, however, are either trivial to falsify or require trust in centralized entities (e.g., search engine caches). This motivates the need for a seamless and standardized internet-wide non-repudiation mechanism, allowing users to share data from news sources, social websites or financial data feeds in a provably secure manner.

Additionally, blockchain oracles that enable data-rich smart contracts typically rely on a trusted third party (e.g., TLSNotary or Intel SGX). A decentralized method to transfer web-based content into a permissionless blockchain without additional trusted third party would allow for smart contract applications to flourish.

In this work, we present TLS-N, the first TLS extension that provides secure non-repudiation and solves both of the mentioned challenges. TLS-N generates non-interactive proofs about the content of a TLS session that can be efficiently verified by third parties and blockchain based smart contracts. As such, TLS-N increases the accountability for content provided on the web and enables a practical and decentralized blockchain oracle for web content. TLS-N is compatible with TLS 1.3 and adds a minor overhead to a typical TLS session. When a proof is generated, parts of the TLS session (e.g., passwords, cookies) can be hidden for privacy reasons, while the remaining content can be verified.

Practical demonstrations can be found at https://tls-n.org/.

1 INTRODUCTION

The overwhelming adoption of TLS [42] for most HTTP traffic has transformed the web into a more confidential and integrity protected communication platform. Despite TLS's adoption, an efficient, secure, privacy-preserving, non-interactive and seamless method to prove communication contents to a third party — i.e. a *standardized method for non-repudiation* — that does not require an additional trusted party is missing.

Such a non-repudiation solution and its proofs would allow more accountability in the web and aid the construction of decentralized blockchain oracles as we outline in the following.

Interestingly, users are currently unable to prove to a third party the content they have observed on a particular website. One of the most popular methods for users to document and share content they watch on the Internet are *screenshots* that are trivial to falsify [21, 30]. A non-repudiation solution would remove the necessary trust towards a user that claims to have observed a given content. Further, currently trusted third parties, such as search engine caches or web archives could add non-repudiable proofs about the content they have observed and thus increase their credibility.

Furthermore, blockchain-based smart contracts [43] can significantly benefit from an efficient non-repudiation solution. If for example a stock market price API provides non-repudiable data, any user could submit verifiably valid stock price information to the blockchain (effectively creating a decentralized blockchain oracle). Because the blockchain-based smart contract verifies the validity of the provided data, peers would only need to trust the data provider, not the peers that actually transmit the data to the blockchain. Generally, this would allow to seamlessly connect real world events with a blockchain and as such enable new application scenarios for smart contracts. Note that existing blockchain oracles either rely on deprecated security protocols (e.g., TLS 1.1 for TLSNotary) or introduce additional trusted third parties (e.g., TLSNotary and Intel SGX).

In this paper, we propose TLS-N, an extension of TLS that enables the seamless integration of non-repudiation between arbitrary parties within TLS. TLS-N allows the generation of privacy-preserving, non-repudiable, non-interactive proofs of the contents of a TLS session. Our solution takes into account the performance requirements of TLS, both in computation and memory to promote adoption and reduce the potential attack surface (e.g. against Denial-of-Service attacks). Our design supports various proof types, that can be shared with other parties, allowing them to verify the conversation contents.

The proof verification requires no additional security assumptions other than those of TLS, and we do not need an additional trusted third party. Note that any non-repudiation solution based on a higher layer (e.g., HTTP), would either require access to the cryptographic TLS keys, violating the layer principle, or would require the deployment and authentication of additional key material, thereby significantly increasing the complexity of the solution.

In TLS-N, by the definition of non-repudiation, message authentication and the identification of at least one TLS peer is guaranteed. We compare TLS-N to existing non-repudiation proposals and identify properties that non-repudiation solutions must possess for particular use cases.

We implement and evaluate TLS-N as an extension of the new TLS 1.3 standard. As such, we implement a TLS-N-enabled web server, web client and an Ethereum-based smart contract that can verify TLS-N proofs.

We find that our prototype implementation incurs an overhead of less than 1.5 milliseconds on existing TLS connections per HTTP request for responses of 10 KB or less, which is a realistic size for an API response. Verifying our proof examples in a smart contract costs between 0.2 and 3 USD due to the currently high gasprice. Prices depend on the proof size and signature type. Note that, once this proof is verified, it can be used by millions of blockchain users.

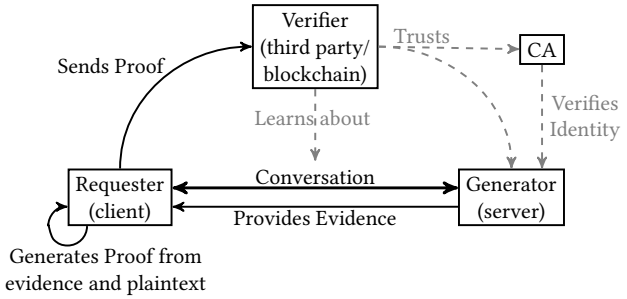


Figure 1: Our view of non-repudiation. First, evidence generation (by generator), second proof generation (by requester), third proof verification (by verifier). Message originator and recipient might act as requester, generator or both.

As a summary, our contributions in this paper are as follows:

- We propose the first secure non-repudiation solution that captures privacy and performance requirements and can be seamlessly integrated with the TLS 1.3 standard [38]. Our solution does not add new security assumptions to those of TLS and does not rely on an additional trusted third party.
- We implement our extension for TLS 1.3 on top of Mozillas NSS library [32] and create an Apache module supporting our extension. Our experimental evaluation shows that a typical proof size as well as the proof generation and verification times grow linear with the size of the data. The server side processing times are low with less than 1ms for 16 KB plaintext without privacy protection and less than 8ms for 16 KB plaintext with privacy protection.
- We provide an Ethereum based smart contract implementation for TLS-N proof verification. TLS-N therefore acts as a practical decentralized blockchain oracle that does not require any additional trusted third party. Users can source data from any TLS-N-enabled content provider, submit it to the blockchain where the smart contract verifies the proof. Note that only the data provider needs to be trusted, and as such any client can submit a TLS-N proof to the smart contract.
- We provide a structured description of non-repudiation properties, possible attacks, requirements and use-cases for non-repudiation solutions.

The remainder of the paper is organized as follows. In Section 2, we define the problem statement and motivate our TLS-based approach before presenting the design of our solution TLS-N in Section 3. In Section 4, we perform its security analysis and evaluate it in Section 5. We overview related work and contrast it to our solution in Section 6, while highlighting attacks on previous TLS-based work. We provide a discussion in Section 7, before concluding the paper in Section 8.

2 PROBLEM STATEMENT

In this section we describe the main problem that we are trying to solve and we discuss relevant use cases and their requirements.

Broadly, we address the problem of non-repudiation in online interactions as seen in Figure 1. Given that such interactions are mainly protected using TLS [11], we focus on the provision of non-repudiation for services that run on top of TLS. TLS is the most widely used security protocol suite on the Internet and provides authentication, confidentiality, and integrity. Although it relies on public-key signatures for authentication, TLS protects message integrity and confidentiality of exchanged messages via shared secret keys that are established at the beginning of the session. Given this, TLS does not provide non-repudiation for the exchanged messages — clearly, a sender of the message can deny having sent the message, given that the Message Authentication Codes have been generated using a shared, symmetric key.

More precisely, we consider the following problem: *Can TLS be extended to provide a compact evidence allowing for efficient proof generation and verification so that the non-interactive proofs allow third parties to verify the TLS conversation contents.* [1]

In addition, since TLS peers might exchange privacy-sensitive content (e.g., login credentials, cookies or access tokens), the TLS extension should provide efficient, privacy-protection features to hide sensitive parts of the conversation from third parties.

Based on previous work in the area, we consider the following non-repudiation types [1, 20, 36, 46]:

Non-repudiation of origin (NRO) provides proof that a message has originated from the specified originator. The evidence is provided by the originator and given the proof, the originator is not able to later deny having sent the message.

Non-repudiation of receipt (NRR) provides proof that a message was received by the specified recipient. The evidence is provided by the recipient and given the proof, the recipient is not able to later deny having received the message.

Non-repudiation of conversation (NRC) provides a proof of a total order of messages sent and received by a party. Intuitively, NRC specifies the conversation and the party's role in it, from the perspective of its system. The specified party is not able to later deny a claim of having sent and received the message in the conversation or the order of messages within the conversation.

Note, that non-repudiation of conversation (NRC) implies non-repudiation of origin (NRO) for all sent messages within the conversation and non-repudiation of receipt (NRR) for all received messages. Therefore, NRC is a stronger proof than NRR or NRO. To highlight the difference between NRO and NRC consider the following example. A web service returns the current stock price for a requested ticker symbol, e.g. for the request `EXAMPLE` the response is $10. Non-repudiation of origin would ensure that the web service answered $10. The answer by itself, however, is not useful without the context of the conversation. Non-repudiation of conversation would ensure that the web service answered $10 after being queried for `EXAMPLE`.

Apart from the non-repudiation type we also consider the following properties of a non-repudiation solution. These properties are motivated by different use cases, as we will show in Table 1.

[1] Here by extended we mean that a proper TLS Extension as specified in [38] can be created.

	Evidence				Privacy		
Use Cases	Non-repudiation Type	Order-Preserving	Request-Response B.	Time	Protecting Sensitive I.	Protection Granularity	Usability
Document Submission	NRR	●	●	●	●	Bytes	●
Public Data Feed	NRC	◐	●	●	-	-	-
Web Archive	NRC	◐	●	●	-	-	-
Misbehaviour in P2P	NRC,NRR	◐	●	◐	◐	Bytes	●

Table 1: For the use cases presented here, non-repudiation of conversation NRC is the most commonly required one. We also find that most use cases require request-response binding and timing information. Additionally, some use cases require privacy protection (e.g., hiding of access tokens or passwords). ● = required property, ◐ = partially required property, - = non-required property.

Order-Preserving: A total order of messages between the TLS peers can be determined based on the proof.

Request-Response Binding: A TLS conversation might include multiple requests and responses. This property ensures a binding between requests and responses based on the proof. This is important as protocols such as HTTP/1.x do not reference the request in the response, e.g. they contain no request ID.

Time: Based on the proof the content creation time (as seen by the peers) can be identified.

Privacy Protecting: Privacy sensitive content (e.g., passwords or cookies) transmitted in a TLS session can be efficiently hidden in the proof.

Possible use cases that would benefit from a non-repudiation solution are (cf. Table 1) (i) Document Submission Systems (e.g., HotCRP) and (ii) Public Data Feeds, e.g. for stock exchange rates and currency exchange rates [35, 44]. Verifiable, public data feeds are essential for the further development and expansion of blockchain-based smart contract applications [43]. Given such a feed, public data can be securely inserted into the blockchain: a smart contract can, on submission of data including a proof, verify the proof and then store the verified data on the blockchain. Any other contract can use such blockchain-based information. This disintermediation removes the need for an additional third party acting as an oracle [26, 45]. Further use cases are (iii) Web Archives [2, 37] for web content or deleted social media content [13], and (iv) proving misbehaviour in P2P networks [15, 33].

2.1 Previous Work and its Limitations

Here, we briefly motivate why existing work is insufficient and motivate our TLS-based design. For a more extensive discussion, please refer to Section 6.

Our design is TLS-based as this comes with multiple key advantages. TLS is ubiquitous. Based on the layer approach, many applications can benefit from a TLS-based solution. TLS provides extension support, allowing for for incremental deployment as our extension is backwards compatible. We can reuse existing, cryptographic primitives of TLS reducing development and maintenance overhead. Additionally, most TLS deployments are based on a few cryptographic libraries simplifying standardization. Finally, TLS already uses an established public-key infrastructure (PKI) necessary for authentication.

Existing TLS-based solutions: Existing TLS-based solutions do not provide secure non-repudiation, as we will show in Section 6.2. In particular, none of the solutions provides NRC. We present attacks against all existing solutions and conclude that none of them has all the required security properties.

(Existing) Application Layer solutions: Non-repudiation can also be managed on the application layer. However, as we will explain in Section 6.2, application layer solutions come with multiple drawbacks. One drawback is that each of the application layer solutions has to provide a separate non-repudiation implementation resulting in many presumably poorly maintained implementations. Therefore, we think that the TLS layer should provide non-repudiation, because it already provides a frequently-used layer offering confidentiality and authentication to all kinds of applications. Furthermore, application layer solutions need their own authentication scheme, while a TLS-based solution can reuse the existing PKI. Finally, in a design as ours, the application layer still retains full flexibility, as it decides what will be included in the proof.

Other existing solutions: There are other solutions providing similar properties, such as TLSnotary [41] and Town Crier [45] that we will discuss in Section 6.3. While TLSnotary only works for older TLS versions and requires trust in a third party, Town Crier is a specific solution for smart contracts that requires special hardware and trust in the attestation service.

Our design, overcomes the shortcomings of previous work, requires no special hardware, no trusted third party and provides a general and portable solution for secure non-repudiation that reuses existing TLS primitives. We describe our design in the following section.

3 OUR DESIGN: TLS-N

Throughout this paper, we consider the following three parties: *(i)* the requester (typically a client machine), *(ii)* the generator (typically a web server) and *(iii)* the verifier (third party or smart contract), as seen in Figure 1. Our design, called TLS-N, provides generator-signed evidence about the TLS conversation to the requester, who can then construct a (redacted) proof. The design is similar to content extraction signatures [39] and redactable signatures [22], which have not been used in combination with TLS. We provide a comparison to these schemes in Section 6.

Figure 2 shows the evidence and proof generation between requester and generator. Initially, they establish a TLS connection and negotiate the TLS-N parameters in the handshake. During the TLS session, the generator keeps a small TLS-N state that is updated using all the sent and received records. This state contains a hash value incorporating all previous records, an ordering vector and a timestamp from the beginning of the session.

Once the requester asks for the evidence, the evidence window that defines which records will be included in the evidence closes.

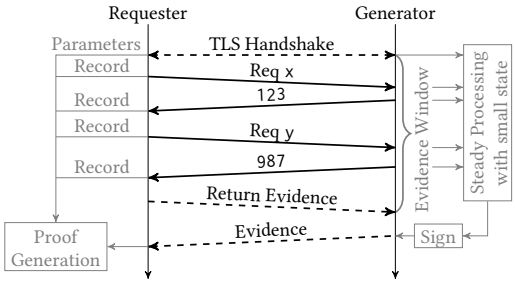


Figure 2: Simplied Overview of TLS-N: The "Return Evidence" message closes the Evidence Window, which determines the evidence-relevant records. The dashed lines represent TLS messages and gray elements represent TLS-N.

Note, that in TLS-N and in contrast to previous work the evidence window begins right after the handshake. To compute the evidence, the generator signs its TLS-N state using its private key. Together with the saved records, this evidence allows the requester to produce non-repudiable proofs for the entire conversation or for a subset of it.

Therefore, the requester retains full control what is included in the proof. To protect sensitive TLS content, the requester can hide entire records or chunks thereof. The generator is oblivious to what the requester considers sensitive and is not involved in the proof generation. By checking the proof, a verifier learns the disclosed content of the TLS session in a non-repudiable manner. We only make standard TLS assumptions, such that both requester and verifier trust the certificates to correctly identify the generator.

3.1 Parameter Negotiation

TLS sessions begin with the handshake during which settings such as the cipher suite are negotiated. If the requester wants to use TLS-N, it includes a TLS-N extension into the handshake. Here, the requester also specifies its preferences for the TLS-N settings. To hide sensitive content, the requester can chose between: *record-level* and *chunk-level granularity*. While chunk-level granularity is more precise it also has a higher computational overhead. In case of chunk-level granularity the requester can also select the chunk size. Again a smaller granularity leads to a higher computational overhead.

Essentially, record-level granularity allows efficient proofs for public data, e.g. in a web archive, or for conversations where entire records can be censored. It represents the most efficient design, as the conversation has to be parsed record-by-record.

The generator can reject or accept the TLS-N settings by including a corresponding response in its handshake message. To ensure that TLS-N cannot be abused for Denial-of-Service attacks, the generator can also enforce the use of a TLS client puzzle [34].

3.2 Evidence Generation

In this Section, we outline how the generator (server) produces the evidence in TLS-N. We discuss the evidence window, the provided evidence and provided auxiliary information to aid proof generation.

In our solution, the evidence collection starts immediately after the TLS handshake. This has two main benefits. One is to prevent Content Omission Attacks (cf. Section 6.2.1) and the other is that TLS-N then does not require an explicit "Collect Evidence" message (proposed by related work [6]). In TLS-N, the evidence window ends as soon the generator receives a "Return Evidence" message.

Order of records

The generation of the evidence is non-trivial as the requester and generator might observe a different order of records. We label the i-th requester and generator records r_i and g_i respectively. If both peers simultaneously send records r_0 and g_0, each peer will observe its sent record before observing its received record, resulting in two different orders: (r_0, g_0) and (g_0, r_0). Note, however, that the two peers have identical partial orders over records generated by one peer, i.e., they observe the same order for all $\{r_i\}$ and for all $\{g_i\}$.

Based on their partial orders, both peers have to agree on a total order. In TLS-N the generator determines the total order of records, as it generates the evidence. To inform the requester about the chosen total order, the generator uses an *ordering vector*. As both peers have the same partial order over $\{r_i\}$ and $\{g_i\}$, the ordering vector is a bit vector encoding the interleaving of $\{r_i\}$ and $\{g_i\}$. In the ordering vector, a 0 corresponds to a record sent by the requester (r_i) and a 1 to a record sent by the generator (g_i). An ordering vector of $(1, 0, 0, 1)$ results in the total record order of (g_0, r_0, r_1, g_1).

Commitments

To allow chunk-level censoring of sensitive information during proof generation, each record of length l_r is split into fixed-sized chunks of the negotiated chunk size l_c. We construct hiding and binding commitments for each of the chunks using a commitment scheme `C()` that takes a chunk and a pseudo-random value, called salt, as input. As the chunk might have low entropy the pseudo-random salt is used to protect the hiding property of the commitment against brute-force attacks.

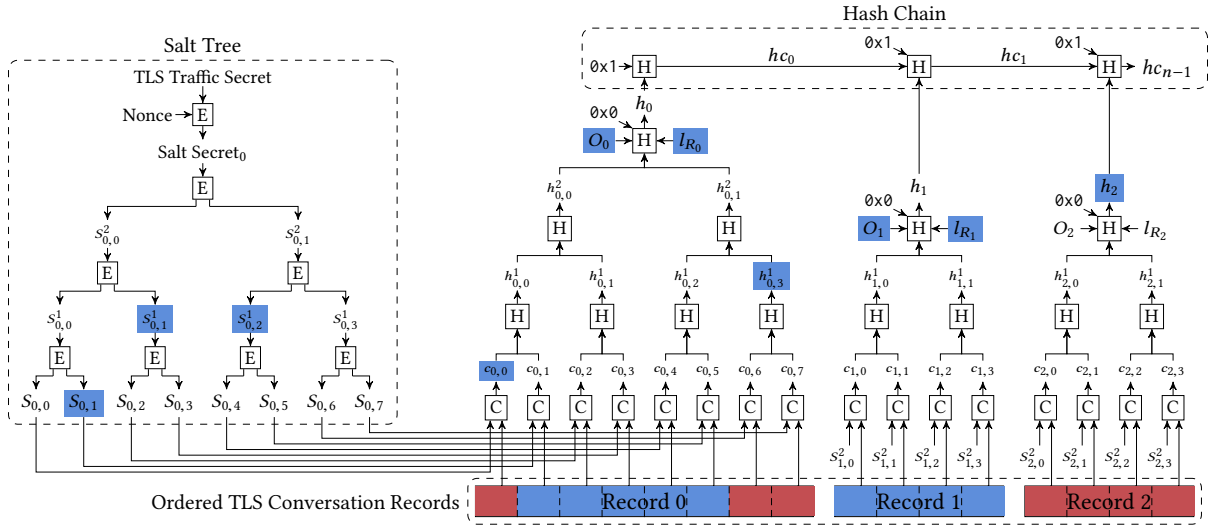
Merkle Tree Generation

To efficiently include commitments in the proof, we construct a Merkle Tree [29] over the commitments, as shown in Figure 3a. The root hashes of the Merkle trees h_i are generated from the children hash values, the length of the record l_r and the originator information O_i. O_i is the i-th element of the ordering vector. We assume that `H()` provides a binding commitment scheme, i.e., is a collision-resistant hash function. To reuse secure, existing TLS primitives we use the hash function negotiated in the cipher suite (typically SHA-256) as `H()`.

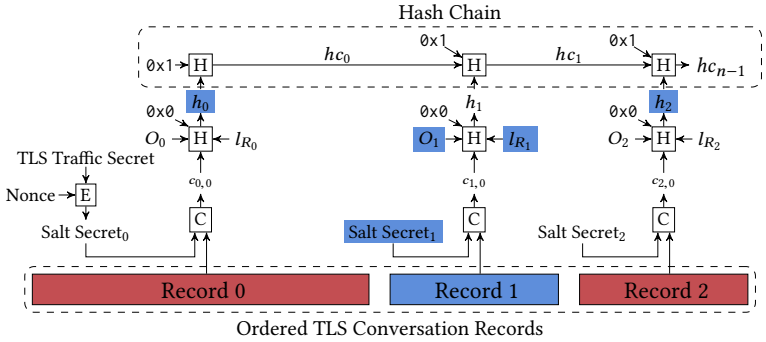
The records' root hashes h_i are combined in a hash chain (hc_i), with hc_{n-1} being the final hash chain state. Using a hash chain ensures a very small storage overhead per TLS session, namely only a single hash value. The hash chain uses markers (`0x0`, `0x1`) to prevent second preimage attacks, as explained in Section 4.3.

Salt Tree Generation

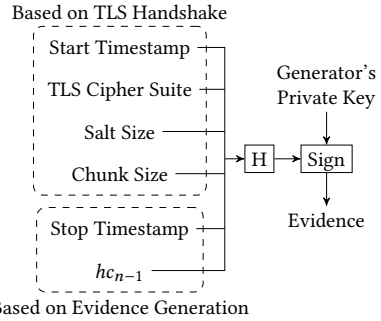
To create hiding commitments using `C()` we need independent, random values S_{i_r, i_c}, called salts, for record i_r and chunk i_c. To preserve hiding, the outputs S_{i_R, i_c} and $S_{i'_R, i'_c}$ have to be independent,



(a) Evidence Generation with chunk-level granularity. The left side shows the salt tree for record 0. Commitments are generated for each chunk using a salt tree leaf (i.e. a salt). To reduce the proof size, a Merkle tree is computed over the commitments. The other salt trees are omitted due to space constraints.



(b) Evidence Generation with record-level granularity. No salt tree or merkle tree computaiton is required. For non-sensitive records, the complete plaintext and the salt secret is included in the proof.



(c) Final step of the evidence generation. Timestamps from the TLS handshake and the evidence generation are included along with the negotiated parameters.

Figure 3: Evidence Generation based on the ordered TLS records using expansion function `E()`, commitment scheme `C()` and hash function `H()`. O_i gives the originator information and l_{R_i} is the record length. The record-based hashes h_i are input to a hash chain, whose result is the final hash value hc_{n-1} that will be signed by the generator. Sensitive content is marked red and is hidden in the proof while all blue elements are included in the proof.

if $i_R \neq i'_R \vee i_c \neq i'_c$. Additionally, to reduce proof sizes we need efficient disclosure of salts for non-sensitive chunks. Therefore, we use a *salt tree* based on the function `E()` to derive the salt values. By using a salt tree, to censor a single chunk, only a logarithmic number of salts need to be revealed in the proof.

The salt tree is computed as follows (cf. Figure 3a): Initially, for each record R_i composed of c chunks, a unique the salt secret is derived from the TLS traffic secret using a record-based nonce. This ensures the generation of a pseudo-random, independent and salt secret as explained in Section 4.3. The derived salt secret is further expanded using `E()` to generate the salt tree. In the salt tree, each output of `E()` is truncated to length $2l_s$ and split into two bitstrings of length l_s, e.g., $S^2_{0,0}$ and $S^2_{0,1}$ in Figure 3a. Until the salt tree has c leaves and thus is large enough to supply a unique salt for every chunk of the record, this process is repeated, i.e. each intermediate secret $S^d_{i,j}$ at depth d is used as an input to `E()` and the output is split again to produce the values $S^{d-1}_{i,2j}$ and $S^{d-1}_{i,2j+1}$. The leaves of this tree are then used as salts. `E()` is a variable-length output pseudorandom function that takes a pseudorandom key, (possibly empty) context information and the output length as inputs. `E()` leaks no information about its key. In TLS 1.3, `HKDF-Expand-Label` is used as `E()` [23, 38].

Chunk-level granularity vs. Record-level granularity

In Figures 3a and 3b we show the overall evidence generation based on the content for chunk-level and record-level granularity. For

chunk-level granularity we combine the salt tree with the commitments, the Merkle tree and the hash chain. On the other hand for record-level commitments, we only generate a single commitment per record and therefore do not need a salt or Merkle tree. In short, for record-level granularity, each record is handled as a single chunk. In both cases we generate the overall hash chain result hc_{n-1} that is subsequently signed.

Providing Trustworthy Timing Information
To provide trustworthy timing information and protect against the Time Shifting attack (cf. Section 6.2.1), our design employs two generator-produced timestamps: one timestamp taken during the TLS handshake and one timestamp taken during evidence generation, i.e. at the beginning and the end of the evidence window. As both timestamps are included in the evidence, the verifier can detect proofs resulting from long TLS sessions and Time Shifting attacks.

As seen in Figure 3c, the evidence consists of the final hash hc_{n-1}, the two timestamps, the chunk size l_c, the salt size l_s and the TLS cipher suite negotiated for this session. When the evidence is requested, it is hashed and signed with the generator's private key. Our design limits the generator's computational overhead as it mostly computes hashes and only provides one signature. The evidence is sent to the requester together with the ordering vector. The requester can use the evidence to construct a variety of different proofs as we will show in the following section.

3.3 Proof Generation And Verification

A central benefit of performing non-repudation over TLS is that we can reuse the already deployed public-key infrastructure (PKI). The signed evidence and its authentication can therefore be verified by third parties. However, third parties only possess the trusted root certificates and miss intermediate certificates required to verify the certificate chain. To allow third-party verification, the requester saves the certificate chain of the TLS connection and includes it in the proof.

For proof generation, the requester uses the n records, the salt secrets, the evidence provided by the generator, the ordering vector, and the certificate chain. Based on these, the requester can generate different kinds of proofs. Here, we give some representative examples.

Proving NRO or NRR: As explained in Section 2, NRC implies non-repudiation of origin (NRO) and non-repudiation of receipt (NRR). Therefore, we can also prove these for one or multiple messages of the conversation. A NRO-proof or NRR-proof for a record i, contains the following: plain text of record i, salt secret i, O_i, hc_{i-1}, $h_{i+1}, \ldots, h_n$, the evidence and the certificate chain.

During proof verification the verifier uses the plain text of record i, its salt secret, the cipher suite and O_i to build the Merkle tree and salt tree, and compute h_i as in Figure 3a. Using the hash chain value hc_{i-1} and the computed h_i the verifier can compute hc_i and using the $h_{i+1}, \ldots, h_n$ the verifier can complete the hash chain and compute hc_{n-1}. Then, the verifier checks the evidence, by verifying the signature using the certificate chain and comparing its hc_{n-1} to the provided hc_{n-1}. Finally, the verifier checks the timestamps based on the application-specific requirements, e.g., testing whether they are too far apart or from a wrong date.

Privacy-preserving, browser-based NRC proof: In this scenario, the browser acts as the requester and a web server as generator. The browser is configured to consider all passwords and cookies sensitive and remove them from the proof, while the web server is unaware of these privacy settings. To hide only the passwords the browser requests evidence with chunk-level granularity. The web server generates the evidence as shown in Figure 3a.

For the proof generation the browser proceeds as follows: For every record i *without* sensitive information, the browser includes its plaintext, its salt secret and O_i. For every record i *with* sensitive information, the browser proceeds as in Figure 3a. All plaintext of all non-sensitive chunks are included with their salts $S_{i,j}$. For chunks with sensitive content the browser includes their commitment. If subsequent chunks are sensitive or non-sensitive the browser includes higher level-nodes from the Merkle tree and the salt tree respectively. Therefore, only $O(\log(c))$ nodes have to be included and the proof size is reduced. Additionally, the proof contains the evidence, and the certificate chain.

During proof verification the verifier uses the proof to re-generate the same evidence as in Figure 3a. For records *without* sensitive content it constructs the Merkle Tree and salt tree, for records *with* sensitive content it constructs the partial Merkle Tree based on the provided plaintext, commitments and hashes. Thereby, the verifier obtains all root hashes h_i, constructs the hash chain and hc_{n-1}. As before the verifier also checks the evidence based on the certificate chain and validates the timestamps.

4 SECURITY ANALYSIS

In this section we present the security analysis of TLS-N. We start by introducing our system and attacker model.

Trust assumptions For the purpose of this paper, we make the following trust assumptions. First, we assume that the used cryptographic primitives such as digital signatures and cryptographic hash functions are secure. We need `H()` to produce a binding and hiding commitment. Note, that the hiding property of hash functions has neither been proven nor rejected. Second, we assume the existence of a Public Key Infrastructure (PKI) that correctly binds entities to the public keys used in TLS, i.e. we inherit the trust assumptions of TLS. Hence, both requester and verifier trust the generator's identity. Third, we assume that private keys used by the generator are not leaked to the adversary and that the generator will not sign arbitrary statements. In any non-repudiation solution relying on digital signatures, incorrect use of the private key compromises the security of the scheme. We consider concrete solutions to the problem of revoked or leaked private keys to be out of the scope of this work (a non-repudiable statement could be included in a blockchain together with a recent Online Certificate Status Protocol (OCSP) response). Finally, the verifier trusts the generator to produce accurate content and timestamps.

4.1 Security Properties

For the security analysis, we adopt the security definition of Content Extraction Signature [39] and match them to our design in

Appendix A. We formally define a record, a conversation and subrelations for each of them. Then, given the design of TLS-N above, we aim to achieve the following security property, some of which are by Steinfeld et al.

Property P_0, is the adapted CES-Unforgeability stating that a valid proof can only be produced for a conversation that is a subconversation of a conversation signed in a proof. Here, we substitute the documents in the definition of CES-Unforgeability with the conversations defined in Appendix A.1.

Property P_1 is the adapted CES-Privacy, stating that a proof leaks no information about hidden parts.

Property P_2: The proof reveals the structure of hidden data. Records with hidden chunks are distinguishable from records without hidden chunks and conversations with missing records are distinguishable from complete conversations.

Property P_3: For every non-hidden record, the originator is known.

Property P_4: The timestamps inside the proof provide tight upper and lower bounds on the generator's time during the conversation.

4.2 Adversarial Model

We assume a computationally-bounded adversary that can take one of two roles. Either the adversary acts as requester trying to generate proofs that lead the verifier to wrong conclusions about the conversation (violating P_0, P_2, P_3 or P_4). Or the adversary acts as a verifier trying to learn hidden data (violating P_1).

Either way, the adversary is allowed to interact with the generator, request evidence for different conversations and inspect proofs published by other users. Furthermore, in accordance to the TLS threat model, on the network the adversary acts as described in the Dolev-Yao Model [12]. In section 6.2.1, we detail attacks on existing solutions under this adversarial model.

4.3 Security Sketch for TLS-N

In this section we provide a brief security analysis of TLS-N. For the full analysis and the proofs, please refer to Appendix A. We go through the different properties and sketch a proof for them.

For Property P_0, the unforgeability of the signature scheme and the collision resistance of `H()`, ensure that the additional data (parameters and timestamps) and the hash chain output are unforgeable. The unforgeability of the hash chain inputs, namely the Merkle hashes, reduces to the collision resistance of `H()`. Given all these, the CES-Unforgeability is satisfied for each records according to the proof provided by Steinfeld et al. [39] as records are almost identical to documents and as the differences are irrelevant for the proof.

For Property P_1 we need to prove that the commitments do not leak any information and that the TLS traffic secret is not revealed, which together with the adversarial network capabilities would disclose hidden data. The hiding property of `C()` is sufficient for the first part given that the salts are pseudorandom and independent. Salts are pseudorandom due to the properties of `E()` and independent as for each record they are derived from an independent salt secret. The TLS traffic secret is not leaked as it is only input to `E()`, which due to its properties does not leak it.

A hidden chunk is observable due to the definition of a record and its length is known due its position, the chunk size and the record size. If the first record of a conversation is not included the proof must start with a hash chain node of the type $\texttt{H(0x1,}hc_{i-1}\texttt{,}h_i\texttt{)}$ instead of $\texttt{H(0x1,}h_0\texttt{)}$, which together satisfies property P_2.

As the records include originator information that is unforgeable due to P_0, P_3 is satisfied. And as the timestamps are likewise unforgeable and are taken at the beginning and the end of the evidence window, tight bounds can be provided on the generator's time, fulfilling P_4.

5 IMPLEMENTATION AND EVALUATION

In this section we describe our TLS-N implementation, its deployment and its evaluation using real-world as well as synthetic traffic.

5.1 Implementation

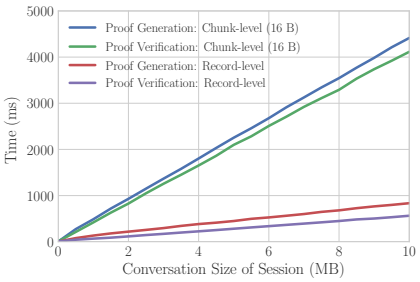
For the purposes of our implementation, we extend the Network Security Services (NSS) library [32] provided by the Mozilla Foundation. We chose the NSS library for its support of TLS 1.3 and because it can be used on the client side, e.g. in Mozilla Firefox, and on the server side, e.g. through the mod_nss Apache module [18, 19]. We implement TLS-N as an extension in NSS and deploy it in a real-world setting using an adapted version of mod_nss and Apache running on an Amazon EC2 node.

We extend TLS so that the requester application can enable the TLS-N extension. The peers negotiate the usage of TLS-N during the handshake. We use a 16-byte salt size, in order to preserve the 128-bit confidentiality protection of TLS [11]. Unless otherwise stated, we also use a 16-byte chunk size, as Figure 4b shows that it provides a good trade-off between granularity and efficiency. For `H()` our implementation uses the hash function of the chosen cipher suite and for `E()` we use the `HKDF-Expand-Label` function with specific labels for salt secret and salt tree generation. `HKDF-Expand-Label` is already used for these properties [38]. As nonce for the salt secret generation, we use the TLS per-record nonce, which is guaranteed to be unique in combination with the traffic secret [38]. For `C()` we use the same function as for `H()`, as we assume that modern hash functions with sufficiently large salts provide a hiding commitment. To reduce the proof size we use TLS certificates using elliptic curve cryptography, namely secp256r1. Overall, we completely reuse cryptographic primitives that are already present in TLS.

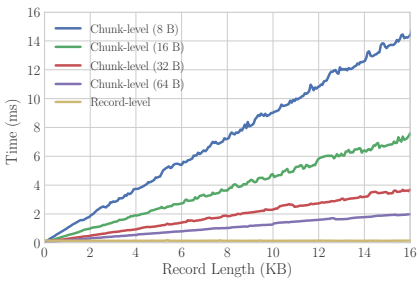
Our extension then constructs a proof according to the settings of the requester application, which provides regular expressions for sensitive content that is then hidden in the proof. Finally, the proof is returned to the requester application. The requester application can store the proof and send it to verifiers. Verifiers can use our library extension to determine the validity of a proof, which includes the necessary salt tree and Merkle tree computations as well as the signature check and the verification of the included certificate chain.

5.2 Blockchain Implementation and Evaluation

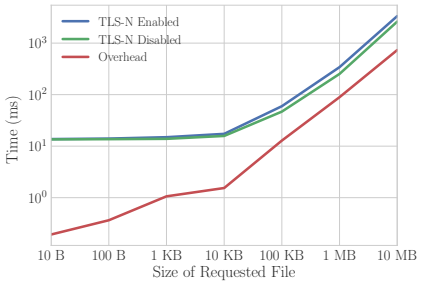
To show that the proof verification can be performed by a blockchain-based smart contract, we provide an Ethereum [43] implementation of the proof verification procedure. The smart contract parses the



(a) Proof generation and proof verification times for random, simulated TLS sessions. Such proofs without hidden data are the worst case for the proof verification. We observe a linear scaling for both times. Each result is averaged over 20 repetitions.



(b) Average processing time for one record depending on its size. Time includes building salt and Merkle tree. Chunk-granularity with sizes ranging from 8 to 64 bytes is compared to record-level granularity. 16 KB is the largest NSS-supported record size.



(c) Average time from sending one HTTP request until the requested file is received with and without our TLS-N extension running on an Apache webserver. We find that TLS-N is usable, even for bigger files. Each result is averaged over 100 measurements.

Figure 4: Performance Evaluation for our implemented extension of TLS-N on client and server side.

proof, computes the salt tree and Merkle tree, and performs a signature verification.

Table 2 shows the respective gas costs in ether and USD (at the time of writing), depending on the conversation size (the cumulative length of all records) and the elliptic curve used in the evidence signature. We also show the basic gas cost that results from the size of a transaction [43]. We show two elliptic curves, because no elliptic curve is supported by both TLS [38] and Ethereum (TLS supports secp256r1 while Ethereum uses secp256k1). The costs differ greatly for the signature schemes, because Ethereum's support for secp256k1 [5]. We had to implement verification for secp256r1 on top of Ethereum, resulting in a verification cost around 1.2 million gas. Overall, we observe, that the proof validation costs are dominated by the basic gas cost and cost for signature verification, whereas our design only adds a marginal cost.

Another issue is the certificate chain verification within the blockchain. To the best of our knowledge there is no blockchain-based system to verify TLS signatures based on the web-PKI. We therefore suggests that the verifying smart contract knows the generator's (e.g. the content provider's) public key so that it can omit the certificate chain verification. Once the smart contract has verified the proof, it knows that the conversation is authentic and can act immediately, e.g. perform a matching payout, save the content or save a content hash in order to avoid future verifications.

Given our smart contract implementation, TLS-N allows to connect web-based content from any TLS-N-enabled content provider such that any smart contract can operate on the provided, non-repudiable data. Note that the requester is not required to be trusted, and as such any requester can submit a TLS-N proof to the smart contract.

5.3 Evaluation

In the following we evaluate the performance of our implemented TLS extension using real world examples and synthetic examples to test its scalability, as shown in Table 3 and Figure 4.

		Conversation Size			
		1 KB		10 KB	
		secp256r1	secp256k1	secp256r1	secp256k1
Costs	Basic Gas	119,758		737,159	
	Total Gas	1,284,723	131,286	1,938,872	782,219
	Ether	0.0257	0.0026	0.0388	0.0156
	USD	2.0381	0.2083	3.0758	1.2409

Table 2: Gas costs for validating public, record-level proofs within our Ethereum smart contract based on the conversation size and the elliptic curve. The basic gas cost is intrinsic for a transaction of that size. Gas and ether prices taken as of May 1st 2017.

5.3.1 Real-world Examples. We evaluate the performance of TLS-N for real-world examples by replaying recorded HTTP connections of web services, such as the Twitter API, Facebook API, YAHOO! API and a Google Search (cf. Table 3). Since the network latency is irrelevant for the proof size and the processing times, we locally replay the recorded traffic between a Lenovo X220 laptop and a server with an Intel Core i7.

We first study the time we deem most critical, the server's processing time during the TLS connection. For conversation sizes below 6 KB the server has a total processing time of less than 3.5 ms. After processing all the records during the connection, the server's the final step of the evidence generation is independent of the conversation size. For chunk-level proofs, we filter all cookies, passwords and authentication tokens, but we also show an unfiltered record-level proof, namely archiving a Wikipedia page, which is significantly more efficient given the conversation size.

5.3.2 Performance Projections. In Figure 4a, we study the scalability of proof generation and verification using synthetically generated proofs. For each size, we create random conversations consisting of 2000-byte records. We observe that the proof generation and verification times scale linearly in the conversation size. Regarding the proof verification, Figure 4a shows the worst-case scenario, as the proofs contain no hidden data and as such all salt and

			Server side during TLS session			Client side, Offline				
			Online	Upon Request						
Use Case	Conversation Size (B)	Number of Records	TLS-N Processing Time (ms)	Evidence Size (B)	Evidence Generation Time (ms)	Proof Type	Proof Size^{+} (B)	Proof Generation Time (ms)	Proof Verification Time (ms)	Hidden Data, e.g., cookies (B)
Twitter API	5,320	3	3.223	84	0.404	Chunk	5,668	9.491	10.345	348
Facebook API	3,187	4	2.041	84	0.394	Chunk	3,629	8.410	9.734	224
YAHOO! API	2,038	4	1.376	84	0.395	Chunk	2,676	8.721	10.032	182
Oanda API	935	2	0.662	84	0.397	Chunk	1,414	6.320	8.767	161
Google Search*	549,530	424	283.162	84	0.398	Chunk	552,180	357.411	231.934	10,001
Wikipedia Archive	585,136	218	11.418	84	0.339	Record	589,924	20.949	20.662	0

Table 3: Use case evaluation: For each use case we give its sizes, total, server-side processing time during the session, the evidence generation time (performed upon request) and the client-side times for proof generation and verification. We highlight the only additional latency during the TLS session. Times are averaged over 20 repetitions. When applicable, the chunk size was 16 B. *The Google Search includes many records due to auto-completion.$^{+}$ The proof size includes the conversation size.

Merkle tree nodes have to be computed. We observe that proofs with record-level granularity are significantly efficient, as Merkle and salt trees only have a single node.

In Figure 4b, we find that server processing times scale linearly in the record size. We plot the average server side processing time for a single record depending on the record length and the chunk size. Bigger chunk sizes require less computation, but have a coarse-grained privacy protection. Along this trend, record-level granularity is by far the most efficient solution.

5.3.3 Latency Overhead. To estimate the real-world overhead of a complete HTTP request, we measure the overhead of our implementation on the latency of HTTP requests to an Apache server running on an Amazon AWS c4.large instance. In each request, the client requests a file of a size between 10 and 10^7 bytes (in powers of 10). For each file size, the average time from sending the request until the file is received is plotted in Figure 4c. Again, we use a chunk size of 16 B. We observe that as long as the whole file can be sent in a single record (i.e. its size is smaller than 16 KB), the latency of TLS 1.3 without TLS-N remains below 10 milliseconds. For larger files the overhead increases but remains below one second for 10MB files. Even though our implementation is neither optimized or parallelized, i.e. the overhead could still be reduced, the overhead appears tolerable. Additionally, recall that this was achieved with a relatively small chunk size of 16 B.

6 SOLUTION SPACE AND RELATED WORK

In the following we summarize existing solutions and their limitations, provide insights on possible strawman solutions and compare their applicability to the use-cases from Section 2 and which properties they satisfy.

6.1 Related Approaches

In this section, we overview related approaches to our design.

Content Extraction Signatures [39] aim to solve a similar problem. Given a signed document, different parts can be extracted while the signature remains valid and is still verifiable by third parties. Content Extraction Signatures consist of a "PseudoRandom Generator with Seed Extraction" corresponding to our salt tree and use merkle trees based on commitments. As they are only designed for a single document, the document length is included in the signature. We include the record length and the originator information in the Merkle root node.

Redactable Signatures [22] as proposed by Johnson et al., are also design-related. Their GGM tree [14] corresponds to our salt tree and they use a Merkle tree, however without commitments. However, they do not include the overall document length in the proof so that a verifier cannot observe how much data was "redacted", if it was "redacted" at the end of the document. Also, their solution requires a marker in every Merkle tree node which is less efficient.

Further additions to redactable signatures provide transparency [7, 24]. These signatures aim to prevent any inference attacks as they hide the structural information of the data. Some schemes even make its impossible for the verifier to observe a redaction. In our work, we intentionally reveal the fact that data was hidden and its structural information (P_2). Previously motivated [22], we provide our motivation for this design through the Content Hiding Attack.

Another related solution, Sanitizable Signatures [8], can be generated by a signer using its private key. They also include the public key of a designated sanitizer and a division into blocks and admissible. These admissible blocks can later be changed by the sanitizer. However, in our design there shouldn't be a designated sanitizer. To simplify adoption and deployment, any generator-accepted peer can act as requester.

Authenticated Data Structures [10, 31] achieve a similar goal as our design. An untrusted party extracts or computes a result based on a signed construct so that the result correctness can be verified by a third party. However, authenticated data structures are more aimed at data outsourcing, e.g. for databases.

6.2 Existing and Strawman TLS Solutions

In this section, we look at other solutions to provide non-repudiation through TLS and present attacks according to our adversarial definition in Section 4. The solutions and their provided properties are summarized in Table 4.

TLS Sign is a proposed extension [16] for TLS 1.1. TLS Sign defines a new sub-protocol (or content type) for TLS called TLSSignOnOff (in addition to the three already existing: Handshake, Application data, and Alert). Both, client and server can use the TLSSignOnOff

Solutions	Evidence				Privacy/Leakage		Usability
	Non-Repudiation	Order-Preserving	Request-Response B.	Time	Protecting Sensitive I.	Protection Granularity	
TLS Sign	NRO	-	-	-	●	Record	◐
TLS Evidence	NRO, NRR	◐	-	◐	●	Record	◐
MAC Chaining	-	-	-	-	-	n/a	-
One Signature	NRC	●	●	●	-	n/a	●
Our Solution	NRC	●	●	●	●	Bytes	●

Table 4: Provided properties satisfied by the different solutions. ● = provided property, ◐ = partially provided property, - = not provided property.

messages to notify their peer that they will start or stop transmitting signed data, i.e., the sub-protocol is used to specify the evidence window. In the evidence window, each record is hashed, and a signature over this hash is generated. When the stop signal is triggered, the generator gathers all hash signature pairs and sends them to the requester as evidence.

The development of TLS Sign had stopped before a final version was released; thus this extension is incomplete. TLS Sign's design presents following disadvantages: TLS Sign is inefficient, because it requires one asymmetric signature per record within the evidence window. TLS Sign is vulnerable to content reordering and content omission attacks. Therefore, TLS Sign only provides NRO and no timing information or message ordering.

TLS Evidence is a TLS extension [6]. Similar to TLS Sign, the client expresses his intent to use TLS Evidence in the TLS extension field. TLS evidence uses a set of new alert messages to be transmitted in the existing alert protocol to define the evidence window. The requester sends an alert message and waits (i.e. he is not allowed to send any messages) for the responding alert. After exchanging these alerts, the evidence window is open until one of the peers sends an alert, triggering a corresponding reply. Then, the peers exchange their certificates and generate the following evidence: a signature over a timestamp, a hash over all sent messages, a hash over all received messages and a hash of the handshake.

TLS evidence has several limitations. First, for human-centered use cases it is unclear when to start and stop the evidence collection.Second, since TLS evidence provides a signature over the hash of all sent and the hash of all received messages, it only provides a partial order within the sent and the received messages. However, the total order between sent and received messages is not preserved. Therefore, TLS Evidence is vulnerable to the content reordering attack, as seen in Figure 5. Because the evidence window can be opened after some content has already been transmitted, TLS Evidence is also vulnerable to the content omission attack. Finally, as the included timestamp is the time of evidence generation, a time shifting attack is possible. Therefore, TLS Evidence only provides NRO, NRR, a partial order and upper time bound.

MAC Chaining was described in the IETF mailing list [25] as combining the already-used Message Authentication Codes (MACs) of individual records to a MAC over the complete communication. MAC Chaining suggests including the MAC of the previous record into the current record and thereby chaining the MAC properties. Finally, to provide the evidence the last MAC of the communication is signed to verify the whole stream with very small overhead. Two variants of MAC Chaining are proposed that either verify only one side or both sides of the communication.

However, in TLS 1.3 a proof for MAC Chaining would have to include the TLS traffic secrets used for authenticated encryption (AEAD) [28] to allow the verification of individual MACs. Given such a proof, including the TLS traffic secrets, the signature of the last MAC and the conversation content, the adversary can create proofs with different conversation content. In short, the unforgeability of P_0 is violated. This is because AEAD authentication tags, for all cipher suites available in TLS 1.3 [38], are not considered collision resistant if the key is known to the adversary. We therefore conclude that MAC Chaining provides *no non-repudiation* as proofs can be forged given an existing proof.

Signing the complete TLS session from the beginning of the handshake until one party closes the connection would be one of the simplest solutions to provide non-repudiation. The evidence window would thereby cover the complete connection. Similar to previous work, such an extension would require the inclusion in the handshake and an additional evidence message at the end of the session.The evidence would be order-preserving. However, this solution requires the requester to store all records in order to be able to compute the final signature and would necessarily result in a big proof size. Finally, such a non-repudiation service offers no privacy protection. In contrast our record-level approach has comparably low computational costs, while being more efficient and providing record-level privacy protection.

Signing content at the application layer could be another non-repudiation solution, as one could argue that such a functionality should not be handled at the TLS layer. Two parties can exchange signed content on the application layer by explicitly requesting to sign data, or employ already existing protocols such as OpenPGP [9]. Application layer non-repudiation however suffers from several disadvantages. First, regarding reusability, an application layer solution would only support a particular protocol/application. Having a TLS layer solution, however, enables any TLS-based application to benefit from non-repudiation. Second, an application layer solution would require that private keys are exposed to the application layer, contradicting the principle of minimum exposure and that the TLS layer is responsible of managing the cryptographic keys.

There are existing solutions providing additional authentication for REST-ful HTTP as studied and extended by Lo Iacono et al. [27]. However, these solutions include different HTTP headers, would have to be extended for future headers and provide authentication only. A TLS-based non-repudiation solution includes all of the HTTP traffic and allows the proof contents to be chosen during proof generation. Finally, in contrast to the existing solutions, TLS-based non-repudiation solves the problem of public key authentication by leveraging the already established web-PKI. Using

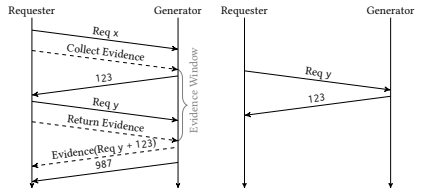


Figure 5: Content Reordering Attack: The left figure shows the original and the right figure the signed conversation. Due to content reordering, the response 123 seems to belong to request y, which is incorrect.

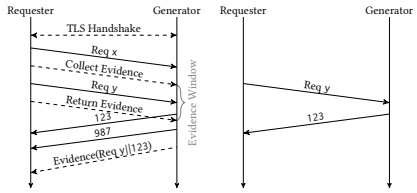


Figure 6: Content Omission Attack: The left figure shows the original and the right figure the signed conversation. Since context is missing in the signed conversation, the response 123 appears to belong to request y which is incorrect.

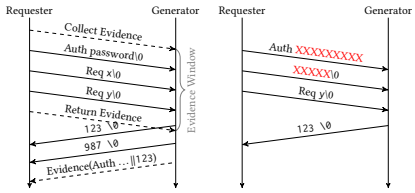


Figure 7: Content Hiding Attack: The left figure shows the original and the right figure the signed conversation. The verifier can not determine whether a long password was used or an additional request took place.

the existing PKI from the application layer would require the exposure of private keys.

As explained in the previous section, earlier works are vulnerable to a number of attacks and do not achieve all of the desirable properties. Furthermore, they don't easily allow the specification of the evidence window and, therefore, limit usability. The protection of sensitive information moreover is only feasible on a record granularity which is impractical for most applications (e.g., stock market API).

6.2.1 Attacks on Existing TLS Solutions. In this section we present attacks against existing TLS solutions apart from the already explained attack on MAC Chaining.

Time Shifting Attack. In Section 2 we described why proofs should contain timing information. However, an adversary acting as requester can manipulate time information included in the proof. The possible kind of manipulation depends the kind of timing information included in the proof.

If a single timestamp is included, the adversary can manipulate the connection according to its type. If the proof generation time is included, the adversary can request the information at time t, then keep the connection open for a time duration Δt and finally request the proof at time $t' = t + \Delta t$. Note that Δt may be substantial as TLS connections can be long lived. Therefore, the proof contains timestamp t' for content requested at time t. Even if Δt is only in the order of minutes, this could have big impacts for data feeds such as stock prices or currency exchange rates. Thus, if such attacks are possible, the timing information is not trustworthy, violating P_4.

Content Reordering Attack. As our system makes no assumptions about the higher level protocol, we must assume that there are cases where the order of messages is important. In particular, the verifier should be able to identify the message order from the perspective of the generator, i.e. the relative order of sent and received messages.

If the adversary can perform a partial content reordering or the content order is not clear from the proof, unforgeability is violated. The scheme cannot prove non-repudiation of conversation (NRC) as the context is unclear. An example relevant for TLS Sign and TLS Evidence is shown in Figure 5.

Content Omission Attack. If the adversary acts as the requester and if the evidence window does not start right after the TLS handshake, i.e., the non-repudiation service allows omission of content as in TLS Sign or TLS Evidence, P_2 is violated. Figure 6 shows a scenario where the adversary requests a resource x before opening the evidence window, immediately requesting another resource y and closing the evidence window. Now, only two records are in the evidence window: the request for y and the response for x.

If the upper-level protocol does not supply any resource identifier in the response, as is the case for HTTP 1.x, it appears to a verifier that x was the legitimate response to the request for y. Therefore such non-repudiation services provide no request-response binding and cannot provide a NRC.

Content Hiding Attack. In this attack the adversary hides important communication content of variable-length protocols by abusing the privacy protection features. In particular, the adversary hides a part of the communication, e.g. a complete request, in order to trick the verifier. An example is shown in Figure 7.

In Figure 7 we assume a simple protocol with three message types: authentication with a password (Auth), requests with an identifier (Req) and responses. As passwords, identifiers and responses can be of variable length, all messages are terminated with a special character (\0).

The adversary first starts evidence collection, then authenticates, sends two requests for resources x and y and then requests the evidence so that only the response for x will be included in the evidence. If the non-repudiation service allows the protection of information, the adversary hides the password along with the request for x (up until the terminating character). The verifier observe the authentication with a hidden password, a request for y and the response 123. The verifier therefore incorrectly assumes that 123 is the correct response for resource y, even though it is 987.

Note that the adversary can send the authentication and the first request within the same TLS record, so that the verifier cannot use TLS metadata to determine whether an additional request was sent.

6.3 Orthogonal Solutions

In the following, we describe orthogonal solutions that offer evidence of TLS sessions using a trusted third party, e.g., TLSnotary [41] and Town Crier [45].

6.3.1 TLSnotary. TLSnotary [41] provides a service that allows a third party auditor to attest a TLS connection between a server and a client. If the client follows a particular protocol with a third party auditor, while initiating a connection to a server, the third

party auditor is able to claim with certainty that the client provided data that originated from the server. TLSnotary modifies the TLS handshake protocol on the client side by leveraging particular properties of TLS 1.0 and TLS 1.1. The modified protocol prevents the client from learning the TLS key material that would allow the client to authenticate traffic from the server. More specifically, the client is not able to generate the server MAC key, only the third party auditor is capable of doing so, effectively preventing the client from crafting traffic that seemingly originates from the server. After the client provided a hash of the traffic, the third-party auditor releases the TLS server MAC key. The client can then verify the message authentication.

TLSnotary Limitations Although TLSnotary provides the capability of notarizing TLS connections, it comes with several limitations and security issues.

First, TLSnotary is only supported up to TLS 1.1. The properties that are used by TLSnotary were removed in versions 1.2 and 1.3. TLS 1.1 and below are considered less secure than current TLS versions. Second, TLSnotary uses and can only use the hash functions MD5 and SHA-1, both of which can be considered deprecated [40]. Third, TLSnotary only supports the RSA key exchange, which does not provide forward secrecy. Last, TLSnotary requires trust in a third party in most use cases, e.g. if the evidence should be publicly verifiable. If the verifier takes the role of the auditor in the protocol, a trusted third party is not required. However, in that case, the verifier needs to take part in the interactive protocol, i.e. evidence of a past session cannot be provided.

6.3.2 Town Crier. Town Crier [45] is a system for authenticated data feeds that leverages the Intel SGX technology to provide publicly verifiable evidence of the contents of a TLS session. It is intended to provide verifiable data feeds for smart contracts (e.g. on the Ethereum blockchain [43]). The core of Town Crier runs in an SGX enclave and can thus provide attestation that the correct code was executed. Town Crier then forwards information that was provided by an HTTPS website to a smart contract on the blockchain.

Town Crier Limitations Similarly to TLSnotary, Town Crier requires a trusted third party, i.e., a client of the service needs to trust Intel since the attestation relies on the security of Intel SGX. In contrast to TLSnotary, Town Crier always requires the trusted third party. In addition, while Town Crier could be modified to provide evidence of TLS sessions in general, it currently only provides data feeds for smart contracts.

7 DISCUSSION

In the following we discuss observations and possible avenues for future work. Our solution is not directly applicable to Datagram TLS (DTLS) that is based on UDP. The DTLS extension remains as a challenge. Moreover, because TLS 1.3 provides simplified resumption features, TLS-N could be extended to support TLS session resumption.

7.1 Validity or Expiry of Proof

A proof should only by considered valid as long as all involved TLS certificates are neither outdated nor revoked. In order to retrospectively understand the time of validity of a proof, either the generator or the validator could make use of a timestamping service attesting the existence of the proof. Besides a centralized service, a cryptographic hash of the proof could also be submitted to a blockchain, effectively timestamping the first occurence of the proof and reducing the trust into a single entity.

7.2 Variable-sized chunking

Our current solution provides fixed-size chunking which is generally applicable, but which might not represent the most efficient solution for the privacy protection of certain applications. Cookies or access tokens (e.g. an OAuth bearer token [17]) are typically stored in the HTTP header. In a hypothetical HTTP mode, TLS-N could support variable-sized chunking, where one chunk could represent one HTTP header. The privacy protection of one header would therefore be more efficient.

7.3 SNARKs for extended Proofs

In some cases, it may be desirable for a requester to provide a more fine grained proof. For example, if a higher-level protocol is used that contains large sections of sensitive variable length data, an attacker could succeed with a content hiding attack (cf. Section 6.2.1). In such cases, the proof will no longer convince a verifier of its validity. Therefore, a requester can extend the TLS-N proof with a zk-SNARK [4]. Such a proof could e.g. prove that the hidden content matches some regular expression, i.e. that no non-sensitive content is censored that is required for the correct semantic meaning of the provided data.

Additionally, a prover can extend a TLS-N proof with a zk-SNARK to prove some statement about the sensitive data. For example, if a party requires proof of sufficient funds, a prover can provide a TLS-N proof of his bank statement but censor his actual bank account balance. He can then provide a zk-SNARK stating that his balance is above some threshold value. Since the TLS-N proof contains a signature of the bank, the verifier is convinced of the origin of the bank account information but since the sensitive content is hidden, he does not receive any unnecessary information.

8 CONCLUSION

In this paper we present TLS-N, the first efficient and privacy-preserving TLS extension that provides non-repudiation of a TLS conversation based on content extraction signatures. Our flexible, parametrized design allows the trade-off between efficiency and privacy, being especially efficient if privacy is not required. No trusted third party or trusted hardware is required while the security assumptions of TLS are inherited and TLS primitives are reused.

Our real-world evaluation including recorded traffic and an Apache Server module demonstrate the usability. For smaller requests, such as API calls, the extra latency is less than 1.5 ms. This secure and efficient non-repudiation solution for TLS will enable parties to provably share the vast amounts of content accessible through TLS — and thus provide disintermediation leading to more trustworthy and decentralized services.

A DETAILED SECURITY ANALYSIS OF TLS-N

Our security analysis of TLS-N is based on the definitions and the security analysis of Content Extraction Signature [39].

A.1 Definitions

In their work, Steinfeld et al. define a document model. Analogously, we define a TLS conversation model. A conversation $\mathbf{c}$ consists of n records, i.e. $\text{len}(\mathbf{c}) = n$, where $\mathbf{c}[i]$ denotes the i-th record. For each record i, $\text{len}(\mathbf{c}[i]) = (l, m)$ where l is the length of the record in bytes and m is the number of chunks. $\mathbf{c}[i][j]$ is the j-th chunk of $\mathbf{c}[i]$ We also adopt the "blank symbol" ?. A chunk is blank if it is hidden in the proof. A record is blank if all of its chunks are hidden and only its merkle root hash is included in the proof. In this case the length is also hidden, i.e. $\mathbf{c}[i] =? \rightarrow len(\mathbf{c}[i]) = (?, ?)$.

Note, however that our records are not quite identical with the documents defined by Steinfeld et al. as the records also contain originator information. $O(\mathbf{c}[i])$ provides the originator information for a given record. If the record is blank, it is blank also, i.e. $\mathbf{c}[i] =? \rightarrow O(\mathbf{c}[i]) =?$. Similarly, each conversation contains additional information $A(\mathbf{c})$, namely the timestamps, the cipher suite, the salt size and the chunk size.

The clear set $\text{Cl}(\mathbf{c}[i]) = \{j \mid \text{len}(\mathbf{c}[i]) = (l, m) \wedge j \leq m \wedge \mathbf{c}[i][j] \neq ?\}$ contains the chunk indices which are not blank for a given record. And for a given conversation $\text{Cl}(\mathbf{c}) = \{i \mid i \leq \text{len}(\mathbf{c}) = n \wedge \mathbf{c}[i] \neq ?\}$ contains the record indices which are not blank.

We say a conversation $\mathbf{c}$ is *complete*, if all records of the conversation are included, i.e. the first hash chain element has the structure $\texttt{H(0x1,}h_0\texttt{)}$, so that h_0 has to be the Merkle hash of the first record.

Steinfeld et al. define a subdocument relation, based on which we define a subrecord and a subconversation relation.

Definition A.1. For any pair of records $\mathbf{r}$ and $\mathbf{r'}$, $\mathbf{r}$ is a subrecord of $\mathbf{r'}$, denoted $\mathbf{r} \leq \mathbf{r'}$, if:

(1) $\text{len}(\mathbf{r}) = \text{len}(\mathbf{r'})$, i.e. they are of equal length and structure
(2) $O(\mathbf{r}) = O(\mathbf{r'})$ and
(3) $\text{Cl}(\mathbf{r}) \subseteq \text{Cl}(\mathbf{r'})$ and
(4) $\mathbf{r}[j] = \mathbf{r'}[j]$ for all $j \in \text{Cl}(\mathbf{r})$).

Definition A.2. For any pair of TLS conversations $\mathbf{c}$ and $\mathbf{c'}$, $\mathbf{c}$ is a subconversation of $\mathbf{c'}$, denoted $\mathbf{c} \leq \mathbf{c'}$, if:

(1) $\text{len}(\mathbf{c}) = \text{len}(\mathbf{c'})$, i.e. the same number of records and
(2) $\mathbf{c}$ and $\mathbf{c'}$ are complete and
(3) $A(\mathbf{c}) = A(\mathbf{c'})$ and
(4) $\text{Cl}(\mathbf{c}) \subseteq \text{Cl}(\mathbf{c'})$ and
(5) $\mathbf{c}[i]$ is a subrecord of $\mathbf{c'}[i]$ for all $i \in \text{Cl}(\mathbf{c})$.

A.2 P_0: Unforgeability

In this section we prove that the adversary cannot produce a forged evidence signature according to our assumptions. In other words, CES-Unforgeability holds for our scheme.

A.2.1 Unforgeability of Signature.

Lemma A.3. *A modified evidence hash leads to an invalid signature that will be detected by the verifier.*

Proof. At the end of the evidence generation the generator signs the evidence hash using its private key. We assume that the private key is handled properly and that it is not used in a signature oracle. Under these conditions, if the adversary can forge a valid signature for a different hash, the unforgeability assumption is violated. Therefore, the signature scheme ensures that only the correct evidence hash has a valid signature. All other signatures will be rejected by the verifier during verification. □

A.2.2 Integrity of Inputs.

In this section, we show that none of the inputs can be modified without leading to a modified evidence hash unless the adversary is able to find a hash collision. We proof this for the operations in reverse order as they are executed.

Lemma A.4. *If any of the final evidence inputs, listed in Figure 3c, is modified the evidence hash will be different.*

Proof. This follows directly from the collision resistance of `H()`. If a modified input would not lead to a different evidence hash, a hash collision would be found. □

Through Lemma A.4 we have established that hc_{n-1} cannot be modified in a valid proof. We go on to show that this implies that none of the hash chain inputs can be modified. Note, that from now on the output length of `H()` (the hash size as part of the cipher suite), the salt size and the chunk size are fixed.

Lemma A.5. *If any of the hash chain inputs for step i with $i > 0$ is modified the final hash chain output hc_{n-1} will be different.*

Proof. Again, this is a direct application of the collision resistance of `H()`. We can apply the collision-resistance argument inductively for all steps. A modification in step i will propagate all the way until the end, as the output of step i, hc_i, is an input to step $i + 1$ unless a hash collision is found. □

Lemma A.6. *If any of the hash chain inputs for step 0 is modified the final hash chain output hc_{n-1} will be different.*

Proof. Again, this is a direct application of the collision resistance of `H()`. □

We have shown that none of the inputs to the hash chain, namely h_i can be modified. The other inputs (`0x1`) are constants. As for each record, the chunk size and its Merkle root hash is now fixed due to the results above, we can use the CES-unforgeability proof of Steinfeld et al. for documents on each record. Note, that records and documents are slightly different. Most importantly, for records the length and the originator information is hashed into the Merkle root node. However, the proof and the arguments about hash collisions apply in the same manner.

Using these lemmas we can prove our theorem about the preserved integrity of TLS records and originator information.

Theorem A.7. *The TLS records, the originator information and the final evidence inputs included in a proof are integrity protected through the evidence signature. Any modification to them will lead to an invalid proof.*

Proof. Any modification of the records or originator information, leads to modified Merkle root hash unless due to the biding property of `C()` and the collision resistance of `H()`.These lead to a modified hash chain output due to Lemmas A.6 and A.5. A different hash chain output or another modified final evidence input leads

to a different evidence hash due to Lemma A.4, which ultimately leads to an invalid proof due to A.3.

Note that this is true for proof with both record-level and chunk-level granularity. In proofs with record-level granularity the Merkle tree consists of a single node and there is only a single commitment. However, the proofs equally apply. □

A.3 P_1 : Confidentiality of Hidden Chunks

The confidentiality of hidden chunks has to be preserved against an adversary even if these chunks have low entropy. To show that the confidentiality is provided we show that:

- The TLS traffic secret is not leaked.
- All disclosed salts are pseudorandom and independent from each other.
- The commitments do not leak information about the hidden chunks.

Lemma A.8. *The TLS traffic secret is not leaked to the adversary through any proof.*

Proof. By its definition, `E()` does not leak the pseudorandom input key. In our design the TLS traffic secret is only fed as an input key to `E()` and never revealed. Therefore, even though proofs might contain the salt secrets, which are derived from the TLS traffic secret, they do not leak the traffic secret to the adversary. □

Lemma A.9. *All salts are pseudorandom values of sufficient length.*

Proof. We assume that the TLS traffic secret is a pseudorandom key, which is a core assumption of TLS [11]. `E()` is defined so that given a pseudorandom input, it will provide a pseudorandom output of a given length. Each salt is generated by alternating calls to `E()` and `Slice()`, e.g. `Slice(E(Slice(E(TLS traffic secret, ctx, 2 · l_S), 0), ctx, 2 · l_S),1)`. The `Slice()` function takes two inputs x, y so that $|x| \geq 2 \cdot l_S$ and l_S is the salt size. `Slice()` is defined as follows:

$$\texttt{Slice}(x,y)=\begin{cases} x[0 : l_S - 1] & \text{if } y = 0 \\ x[l_S : 2 \cdot l_S - 1] & \text{if } y = 1 \end{cases}$$

Clearly, `Slice()` has a pseudorandom output given a pseudorandom input, as it only selects a part of the input.

Therefore, each function in the call chain generates pseudorandom output of size $\geq l_S$ and thereby all salts are pseudorandom values with sufficient length as the salt size l_S is chosen as a security parameter.

Note, that repeated application of `E()` might increase the attacker's distinguishing advantage. However, in our design the salt tree can have at most 17 levels and therefore salts are the result of at most 17 consecutive calls to `E()`. Therefore, salts are still pseudorandom, as an attacker could otherwise use this constant overhead in a distinguishing attack. Our salt tree has at most 17 levels as in the worst case, a record would have $2^{16} - 1$ bytes and the chunksize would be 1 byte leading to 2^{16} leaf nodes. □

Lemma A.10. *The adversary cannot distinguish the values of two neighbouring nodes inside the salt tree from two random values of the same length.*

Proof. Two neighbouring nodes S_x, S_y are the result of S_x = `Slice(E(`K_p`, ctx, 2 · l_S), 0)` and S_y = `Slice(E(`K_p`, ctx, 2 · l_S), 1)` based on a parent node K_p. We prove their independence using a reduction to the pseudorandomness property of `E()`.

Assuming the existence of a polynomial-time adversary $\mathcal{A}$, able to determine S_x, S_y from two random values of the same length with non-negligible probability, we can distinguish `E()` from a random oracle [3] as follows.

(1) Query the oracle Fn() using a random x for length $2 * l_S$
(2) Compute S_x = `Slice(Fn(x), 0)`, S_y = `Slice(Fn(x), 1)`
(3) Query $\mathcal{A}$ using S_x, S_y
(4) If $\mathcal{A}$ detects dependent values, output 1
(5) Else, output 0

If the adversary $\mathcal{A}$ detects dependent values, they must be generated by `E()`, hence the algorithm outputs 1 to signal that it is in the real world. Thereby, the polynomial-time algorithm has a significant advantage.

□

Lemma A.11. *Two salt values that are included in (potentially different) proofs are independent from each other. In particular the adversary obtains no information about a salt $S_{i,j}$ from a salt of a different session, a different record (e.g. $S_{i',j}$) or a different chunk (e.g. $S_{i,j'}$).*

Proof. Given a salt from a different session, the adversary can not obtain any information about $S_{i,j}$, because they are derived from independent traffic secrets. By the TLS design [11], each session has a fresh traffic secret that is independent from previous traffic secrets.

Given a salt from a different record inside the same session, the adversary obtains no information about a salt $S_{i,j}$ because they are derived from independent salt secrets. We prove this using a case separation: Either the two salt secrets are based on different traffic secrets, as traffic secrets can change [38]. In that case the derived salts are independent. Otherwise the nonce is different for the different. In this case, `E()` is designed to generate independent outputs even for correlated inputs, such as nonces, given a pseudorandom key (the traffic secret) [23].

Given a salt from the same record but for a different chunk $S_{i,j'}$, the adversary obtains no information about $S_{i,j}$. As `E()` does not leak its input key, the adversary cannot compute a parent node of $S_{i,j'}$ that could be used to derive $S_{i,j}$. What is left to prove is that there is no dependency between $S_{i,j}$ and $S_{i,j'}$. If $S_{i,j}$ and $S_{i,j'}$ are direct neighbours, Lemma A.10 proves the independence. Otherwise they have to have respective parent nodes that are direct neighbours. These parent nodes are independent according to Lemma A.10 and therefore the derived values are independent. The discussed parent nodes exist in this case, as both nodes are from the same tree but not from the same branch and therefore must have parent nodes that are direct neighbours. They cannot be from the same branch as no two salts from the same branch are included into a proof given that one discloses the other and only a minimal number of salts is included in the proof.

□

Lemma A.12. *A commitment leaks no information about an underlying chunk to an efficient adversary.*

Proof. A chunk might have low entropy. In the worst case the entropy of the chunk might be a single bit. Therefore, our commitment scheme uses the salts to provide the hiding property. As Lemma A.9 shows, these salts are pseudorandom. Additionally, as A.11 shows, the salts are independent, so that the adversary cannot learn about a salt from other salts in the proof or different proofs. Using these properties, the commitment scheme `C()` provides the desired property as we assume that it is hiding. □

Theorem A.13. *No information is leaked about sensitive chunks that are hidden in a proof.*

Proof. The adversary might obtain information about sensitive chunks in different ways. Given its network capabilities, the adversary can capture the encrypted TLS traffic and decrypt later given the TLS traffic secret. However, Lemma A.8 shows that the traffic is not leaked by the proofs. Alternatively, the adversary could try to derive information about sensitive chunks given the commitments contained in a proof. However, Lemma A.12 shows that the commitments leak no information about the underlying sensitive chunks. □

A.4 P_2: Structure of Hidden Data

In this section we show the observable structure of hidden data.

A.4.1 Structure of Hidden Chunks. As the chunk size and the record length are unforgeable the Merkle tree structure is fixed. Therefore, no second-preimage attacks on the Merkle tree are possible and whenever a chunk is hidden, a Merkle tree node is inserted into the proof. The Merkle tree node reveals the index of the hidden chunk and together with the chunk size and record length also its size.

A.4.2 Incomplete Conversations. For the conversation in Figure 3a, a proof might simply include hc_1, Salt Secret$_2$ and record 2. (For this example we ignore data sensitivity as shown in the Figure.) The proof can be verified by computing hc_{n-1} and verifying the signature. However, the conversation is incomplete as it is unclear how many records were part of the conversation. This incompleteness is clearly observable due to the inclusion of the hash chain element hc_1 in the proof.

REFERENCES

[1] ISO/IEC JTC 1. 2009. ISO/IEC 13888-1:2009. (2009). Information technology - Security techniques - Non-repudiation - Part 1: General.

[2] Internet Archive. 2017. Wayback Machine. https://archive.org/web/. (2017). Accessed: 2017-05-19.

[3] Mihir Bellare and Phillip Rogaway. 2005. *Introduction to Modern Cryptography.*

[4] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. *SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge.* Springer Berlin Heidelberg, Berlin, Heidelberg, 90–108. https://doi.org/10.1007/978-3-642-40084-1_6

[5] Alex Beregszaszi. 2016. Ethereum ECVerify. (2016). https://gist.github.com/axic/5b33912c6f61ae6fd96d6c4a47afde6d.

[6] Mark Brown and Russ Housley. 2006. *Transport Layer Security (TLS) Evidence Extensions.* Internet-Draft draft-housley-evidence-extns-01. IETF Secretariat. https://tools.ietf.org/html/draft-housley-evidence-extns-01 https://tools.ietf.org/html/draft-housley-evidence-extns-01.

[7] Christina Brzuska, Heike Busch, Oezguer Dagdelen, Marc Fischlin, Martin Franz, Stefan Katzenbeisser, Mark Manulis, Cristina Onete, Andreas Peter, Bertram Poettering, and Dominique Schröder. 2010. Redactable Signatures for Tree-structured Data: Definitions and Constructions. In *Proceedings of the 8th International Conference on Applied Cryptography and Network Security (ACNS'10).* Springer-Verlag, Berlin, Heidelberg, 87–104. http://dl.acm.org/citation.cfm?id=1894302.1894310

[8] Christina Brzuska, Marc Fischlin, Tobias Freudenreich, Anja Lehmann, Marcus Page, Jakob Schelbert, Dominique Schröder, and Florian Volk. 2009. *Security of Sanitizable Signatures Revisited.* Springer Berlin Heidelberg, Berlin, Heidelberg, 317–336. https://doi.org/10.1007/978-3-642-00468-1_18

[9] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. 2007. *OpenPGP message format.* Technical Report.

[10] Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, and Stuart G. Stubblebine. 2001. Authentic Third-party Data Publication. In *Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security: Data and Application Security, Development and Directions.* Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 101–112. http://dl.acm.org/citation.cfm?id=646118.758638

[11] Tim Dierks. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. (Aug. 2008). https://doi.org/10.17487/rfc5246

[12] D. Dolev and A. Yao. 2006. On the Security of Public Key Protocols. *IEEE Trans. Inf. Theor.* 29, 2 (Sept. 2006), 198–208. https://doi.org/10.1109/TIT.1983.1056650

[13] Marina Fang. 2017. Former Trump Adviser Roger Stone Admits Collusion With WikiLeaks, Then Deletes It. (2017). Available from: http://www.huffingtonpost.com/entry/roger-stone-donald-trump-julian-assange_us_58bc24cae4b0d2821b4ec16c.

[14] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to Construct Random Functions. *J. ACM* 33, 4 (Aug. 1986), 792–807. https://doi.org/10.1145/6490.6503

[15] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical accountability for distributed systems. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 175–188.

[16] Ibrahim Hajjeh and Mohamad Badra. 2007. *TLS Sign.* Internet-Draft draft-hajjeh-tls-sign-04. IETF Secretariat. https://tools.ietf.org/html/draft-hajjeh-tls-sign-04 https://tools.ietf.org/html/draft-hajjeh-tls-sign-04.

[17] D. Hardt. 2012. *The OAuth 2.0 Authorization Framework.* RFC 6749. RFC Editor. http://www.rfc-editor.org/rfc/rfc6749.txt http://www.rfc-editor.org/rfc/rfc6749.txt.

[18] Christian Heimes. 2017. mod_nss. (2017). Available from: https://pagure.io/mod_nss.

[19] Christian Heimes. 2017. mod_nss branch TLS 1.3. (2017). https://pagure.io/fork/cheimes/mod_nss/branch/tls13.

[20] Siegfried Herda. 1995. Non-repudiation: Constituting evidence and proof in digital cooperation. *Computer Standards & Interfaces* 17, 1 (1995), 69 – 79. https://doi.org/10.1016/0920-5489(94)00044-H

[21] Simitator.com Social Imitator. 2017. Fake Twitter Tweet. (2017). http://simitator.com/generator/twitter/tweet.

[22] Robert Johnson, David Molnar, Dawn Xiaodong Song, and David Wagner. 2002. Homomorphic Signature Schemes. In *Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology (CT-RSA '02).* Springer-Verlag, London, UK, UK, 244–262. http://dl.acm.org/citation.cfm?id=646140.680938

[23] Hugo Krawczyk. 2010. *Cryptographic Extraction and Key Derivation: The HKDF Scheme.* Springer Berlin Heidelberg, Berlin, Heidelberg, 631–648. https://doi.org/10.1007/978-3-642-14623-7_34

[24] Ashish Kundu and Elisa Bertino. 2008. Structural Signatures for Tree Data Structures. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 138–150. https://doi.org/10.14778/1453856.1453876

[25] Sergio Lerner. 2015. Renegotiation and TLSNotary. IETF TLS mailing list. (2015). Available from: https://mailarchive.ietf.org/arch/msg/tls/8ZK37bSWPvHf0p0X9nFzVAoT-04.

[26] Oraclize Limited. 2017. Oraclize - blockchain oracle service, enabling data-rich smart contracts. http://www.oraclize.it. (2017). Accessed: 2017-05-19.

[27] Luigi Lo Iacono and Hoai Viet Nguyen. 2015. *Authentication Scheme for REST.* Springer International Publishing, Cham, 113–128. https://doi.org/10.1007/978-3-319-19210-9_8

[28] D. McGrew. 2008. *An Interface and Algorithms for Authenticated Encryption.* RFC 5116. RFC Editor. http://www.rfc-editor.org/rfc/rfc5116.txt http://www.rfc-editor.org/rfc/rfc5116.txt.

[29] Ralph C. Merkle. 1988. *A Digital Signature Based on a Conventional Encryption Function.*

[30] Gianluca Mezzofiore. 2017. All the reasons why that James Comey's 'pee tape' tweet is fake. (2017). http://mashable.com/2017/05/10/james-comey-fake-tweet-pee-tape-debunk/.

[31] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated Data Structures, Generically. *SIGPLAN Not.* 49, 1 (Jan. 2014), 411–423. https://doi.org/10.1145/2578855.2535851

[32] Mozilla. 2017. Network Security Services. (2017). Available from: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

[33] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[34] Erik Nygren, Samuel Erb, Alex Biryukov, Dmitry Khovratovich, and Ari Juels. 2016. *TLS Client Puzzles Extension.* Technical Report. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-nygren-tls-client-puzzles-02

[35] OANDA. 2017. Exchange Rate API for Businesses and Corporates. https://www.oanda.com/solutions-for-business/feed.html. (2017). Accessed: 2017-05-19.
[36] Jose Antonio Onieva, Javier Lopez, and Jianying Zhou. 2009. *Secure Multi-Party Non-Repudiation Protocols and Applications*. Advances in Information Security, Vol. 43. Springer. https://doi.org/10.1007/978-0-387-75630-1
[37] Bill Palmer. 2016. Did Bernie Sanders really dare Trump to send protesters? Its not on Twitter. Deleted or hoax? (2016). Available from: http://bit.ly/1MgB9Ol.
[38] Eric Rescorla. 2016. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-tls13-18. IETF Secretariat. https://tools.ietf.org/html/draft-ietf-tls-tls13-18 https://tools.ietf.org/html/draft-ietf-tls-tls13-18.
[39] Ron Steinfeld, Laurence Bull, and Yuliang Zheng. 2002. Content Extraction Signatures. In *Proceedings of the 4th International Conference Seoul on Information Security and Cryptology (ICISC '01)*. Springer-Verlag, London, UK, UK, 285–304. http://dl.acm.org/citation.cfm?id=646283.687991
[40] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The first collision for full SHA-1. (2017). https://shattered.io/static/shattered.pdf.
[41] TLSnotary. 2014. TLSnotary - a mechanism for independently audited https sessions. (2014). Available from: https://tlsnotary.org/TLSNotary.pdf.
[42] Trustworthy Internet Movement. 2017. SSL Pulse. (2017). Available from: https://www.trustworthyinternet.org/ssl-pulse/.
[43] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).
[44] XE. 2017. XE Currency Data Feed. http://www.xe.com/datafeed/. (2017). Accessed: 2017-05-19.
[45] 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 (CCS '16)*. ACM, New York, NY, USA, 270–282. https://doi.org/10.1145/2976749.2978326
[46] Jianying Zhou and Dieter Gollmann. 1997. Evidence and non-repudiation. *Journal of Network and Computer Applications* 20, 3 (1997), 267 – 281. https://doi.org/10.1006/jnca.1997.0056