ROSEN: RObust and SElective Non-repudiation (for TLS)

Srdjan Čapkun
srdjan.capkun@inf.ethz.ch
ETH Zurich
Switzerland

Gene Tsudik
gene.tsudik@uci.edu
UC Irvine
USA

Ercan Ozturk
ercano@uci.edu
UC Irvine
USA

Karl Wüst
karl.wuest@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT

A versatile non-repudiation service that can be used directly and without application-specific modifications is desirable in many realistic use-cases. Since TLS is one of the most popular current means of secure communication, several proposals have been made for augmenting it with a general, flexible and efficient non-repudiation service. However, none of them offers sufficient robustness for scenarios that require high reliability. Also, they lack flexibility by requiring the party providing non-repudiable evidence to do so for all content transmitted within a given TLS session. In this paper, we propose ROSEN, an extension for TLS that provides non-repudiation using an efficient checkpointing mechanism that minimizes loss of evidence in the presence of faults in order to increase robustness and ensure reliability. In addition, ROSEN inherits privacy-preserving properties of prior methods and introduces *selective non-repudiation* which allows the party providing non-repudiable evidence to selectively and efficiently redact parts of the session so as to make them repudiable.

CCS CONCEPTS

• Security and privacy → Cryptography; Authentication; Privacy-preserving protocols; Web protocol security.

KEYWORDS

non-repudiation, TLS, robustness, privacy

ACM Reference Format:

Srdjan Čapkun, Ercan Ozturk, Gene Tsudik, and Karl Wüst. 2021. ROSEN: RObust and SElective Non-repudiation (for TLS). In *Proceedings of the 2021 Cloud Computing Security Workshop (CCSW '21), November 15, 2021, Virtual Event, Republic of Korea*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3474123.3486763>

1 INTRODUCTION

In recent years, TLS [11] has been very widely adopted, to the extent that a majority of websites are now browsed using HTTPS, at least by Chrome users [20]. However, TLS does not provide a

secure means of demonstrating origin authenticity and integrity of content communicated between a client and a server to third parties, i.e., TLS currently has no support for non-repudiation. However, a non-repudiation option would be valuable for many use-cases, such as authenticated web archives and provable banking interactions.

While several techniques for augmenting TLS with non-repudiation have been proposed, none is widely deployed and many have severe flaws (see Section 3.1). The most promising approach is TLS-N [37]. It offers efficient non-repudiation of a one-way authenticated TLS session and allows after-the-fact redactions of signed session data to keep selected data confidential from a verifier. In TLS-N, a client typically acts as a *requester* of non-repudiation evidence over the TLS session whereas the server assumes the role of a *generator*.

Although one-way (server) authentication in TLS is prevalent in the context of web browsing, two-way (client or mutual) authentication is prominent for TLS usage in corporate settings. For example, Cloudflare offers mutually-authenticated TLS connections [8] for certain environments (e.g., IoT), while both Facebook and Google employ mutual-authentication in their data centers [2, 6]. Also, in the financial industry, Visa enforces mutual authentication for payments [1]. An effective TLS non-repudiation technique for these environments could reduce distrust between networks and provide an undeniable proof of the session to communicating parties. Yet, to the best of our knowledge, there is no non-repudiation mechanism for TLS that can be used out-of-the-box in these security- and reliability-critical settings.

Security concerns in these settings include adversaries that (1) mount denial-of-service attacks, (2) avoid providing non-repudiation evidence, and (3) deceive other parties into generating false evidence within a conversation. Another major concern is reliability, or equivalently, robustness. Although there are numerous methods to recover offline data in case of faults, current non-repudiation techniques lack mechanisms that prevent such attacks or that detect them in real time.

Unfortunately, reliability is a concern not unique to mutual authentication. It also applies to vanilla one-way-authenticated TLS settings. Unreliability of web-based applications, such as online banking, is exemplified with familiar warnings, such as: "Do not close this window!" or "Do not press the back button!". A mis-click during a web-based banking transaction could cause serious account errors and wreak havoc on a client's finances. Unfortunately, clients have no way to later securely demonstrate evidence of a failed transaction or session.

This work is licensed under a Creative Commons Attribution International 4.0 License.

CCSW '21, November 15, 2021, Virtual Event, Republic of Korea

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8653-1/21/11.

<https://doi.org/10.1145/3474123.3486763>

Motivated by the above, we propose a robust non-repudiation technique, called ROSEN: **RO**bst and **SE**lective Non-Repudiation that is suitable for both one-way and mutual authentication settings – the latter is referred to as symmetric non-repudiation. In addition to increased robustness, ROSEN removes the implicit “all-or-nothing” assumption of prior non-repudiation methods – that one party provides non-repudiation for everything within a session. We believe that this model hinders adoption, since most parties would not want to provide non-repudiation for **all** communication within a given TLS session. Instead, ROSEN allows parties to choose the exact parts of the session to be non-repudiable via flexible redactions, thus achieving *selective non-repudiation*.

ROSEN has an advantage over prior techniques by providing greater *robustness* through use of an efficient checkpointing mechanism, which only stores minimal state between checkpoints and shares non-repudiation evidence at pre-determined intervals. This reduces loss of evidence in case one party aborts the TLS session.

Also, both parties in ROSEN (client and server) can act as a requester and generator, as well as an evidence verifier. Notably, the generator role involves periodic evidence generation. An evidence verifier confirms the correctness of received evidence during the session.

To summarize, this paper makes the following contributions:

- We propose ROSEN that improves on current non-repudiation methods to provide a robust non-repudiation mechanism for TLS. ROSEN supports selective non-repudiation and provides increased robustness through the use of an efficient checkpointing mechanism.
- We implement ROSEN as a TLS extension in Mozilla’s Network Security Services (NSS) [10] library.
- We analyse ROSEN, evaluate its performance and show that its overhead is sufficiently low for real-world use.

Organization. Section 2 introduces some of the primitives used by our design and Section 3 overviews the problem space (with further related work in Section 6). Our technique, ROSEN is presented in Section 4 and its implementation details and evaluation (latency and security) results are shown in Section 5. We discuss possible extensions and improvements in Section 7 and conclude the paper in Section 8.

2 BACKGROUND

In this section, we briefly describe cryptographic primitives used in ROSEN. These include commitments, Merkle hash trees and hash chains. Given the familiarity of the reader with these concepts, this section can be safely skipped.

2.1 Commitments

A commitment [16] is a cryptographic primitive with “hiding” and “binding” properties that allows committing to a value – during the *commit phase*, with the option of revealing the value at a later time – namely, at *reveal phase*. The “hiding” property ensures that the value is hidden by the commitment and the “binding” property ensures that the value cannot be changed – unless the commitment is changed.

In this paper, we leverage hash-based commitments in which a cryptographic hash function is applied to data with additional randomness – a salt (See Figure 1). A sizable salt hides the committed data by preventing brute-force attacks.

2.2 Merkle (Hash) Trees

A Merkle tree [33] is a tree in which leaf nodes hold data (i.e., *data node*) and non-leaf nodes are the hash of (the concatenated values of) its children (See Figure 1).

An inclusion proof for a data node consists of the siblings of the nodes on the path from the data to the tree root. To check that some data was included in the Merkle tree, a verifier uses this inclusion proof to recompute the value of the root of the Merkle tree and checks equality with the known root value. Since cryptographic hash functions are collision resistant, a proof of inclusion convinces a verifier that specific data are included in a Merkle tree. Maximum number of nodes required for an inclusion proof is bounded by the log of the depth of Merkle tree.

Figure 1: A commitment-based Merkle hash tree with four data $D_{[0,3]}$. The inclusion proof for D_0 consists of H_1 and H_5 . $H()$ is a cryptographic hash function and $H_{[0,5]}$ are hash-based commitments with random salts $r_{[0,3]}$. R is the root of the tree.

2.3 Hash Chain

A hash chain [27] is produced by applying a cryptographic hash function on a piece of data – possibly with additional data (See Figure 2). The head of a hash chain refers to the result of the last application of the hash function whereas the tail refers to the initial state of the chain. If additional data is hashed at each successive application, the additional data at each application comprises the inclusion proof. The inclusion proof is bounded by the number of hash function application.

3 PROBLEM STATEMENT

We now discuss TLS non-repudiation, overview prior techniques and present some use-cases that motivate symmetry and robustness. Then, we derive requirements and define desired properties for a new non-repudiation technique. We consider three types of non-repudiation (NR) [37]:

Non-repudiation of origin (NRO): proof that a message was sent by the specified sender.

Figure 2: A hash chain with additional data $A_{[0,3]}$ included at each successive application of H – a cryptographic hash function. Initial tail value $H_0 = H(A_0)$, head of the hash chain is H_3 .

Non-repudiation of receipt (NRR): proof that a message was received by the specified recipient.

Non-repudiation of conversation (NRC): proof of a total order of messages that were sent and received by a specified party.

Trust Assumptions. We assume a global Public Key Infrastructure (PKI) that allows a TLS server and a client to verify each other’s identities, as required by the TLS session, i.e., one-way or mutual authentication. In conjunction with this, we assume entity private keys are not leaked, therefore produced signatures are indeed generated by the entities identified by corresponding public keys. We assume that each NR proof is accompanied by the certificate chain of the proof generator, allowing the proof verifier to validate the origin of that proof.

Furthermore, we assume that the proof verifier correctly verifies all signatures (on proofs and certificates) and checks that each relevant certificate was/is valid at the time of evidence generation/verification.¹

In terms of the underlying cryptographic primitives, we assume they are correctly implemented and verified. For example, hash function-based commitments are “hiding” and “binding” and key derivation functions (KDFs) do not leak any information about their input. Put another way, attacks against completeness and soundness of cryptographic primitives are out of scope of this paper.

3.1 Prior Work & Limitations

TLS Sign [4] was an incomplete proposal for a TLS extension that provides a signature over each TLS record. Since the records are signed individually, they can be reordered arbitrarily, i.e. TLS Sign provides only NRO.

TLS Evidence [5] is a TLS extension that provides a signature over a hash of all sent, and a hash of all received, messages within an evidence window. This yields only partial message order: since sent and received messages are ordered separately, they can be interleaved arbitrarily. Thus, TLS Evidence does not provide NRC.

MAC chaining [30] was proposed on the IETF mailing list as a mechanism to use MACs of TLS records to provide a MAC over the whole session, which can then be signed. The idea was that this way no additional hash over the session has to be computed and MAC chaining can happen in a streaming manner, i.e. the data does not need to be stored for a longer time. However, verifying MACs requires the corresponding key, which would allow an attacker to change messages, since for all cipher suites available in TLS

1.3, authentication tags are not collision resistant if the attacker knows the key. Therefore MAC chaining does not actually provide non-repudiation.

TLS-N [37] provides NRO, NRR, as well as NRC for a TLS session, i.e., a TLS party can prove to a third party that a TLS session occurred along with its contents. Notably, TLS-N preserves the message order observed by the signing party. TLS-N also allows the evidence recipient to redact messages, in order to remove sensitive data, such as passwords or cookies, when forwarding the evidence to third parties. However, TLS-N does not allow for the reverse, i.e., the signing party cannot decide to exclude parts of the session from the proof. Message redaction in TLS-N focuses exclusively on *confidentiality* of redacted content.

As an alternative to integrating it into TLS, non-repudiation could also be implemented at the application layer. However, this would require investing a significant effort for each application, i.e., design and implementation costs. Also, interoperability problems would likely arise due to incompatible implementations. Therefore, we argue that non-repudiation ought to be provided at TLS layer since it is widely deployed and interoperable.

Preview of Goals. We believe that prior TLS-level non-repudiation techniques need improvement in two aspects: (1) support *selective non-repudiation* that allows the signing party to choose the exact parts of a session that *should not* be repudiable, and (2) increase robustness – minimize loss for both parties if the session is interrupted, (e.g., due to a failure or malicious behavior) i.e., non-repudiation must cover as much of the session as possible. We discuss this in detail in Section 3.3 below.

3.2 Motivating Applications

Below, we list a few applications that could greatly benefit from symmetric and/or robust non-repudiation features.

Security- and Reliability-Critical Networks. The use of client authentication in bank networks and data centers [1, 2, 6] stems from the criticality of these systems. A successful attack against a data center or a banking network may result in a loss of millions of dollars, or in major privacy leaks. Reliable tracking of interactions between machines in these settings allows for identifying problems and preventing them in the future. If used between mutually distrustful networks (e.g., different banks processing transactions), a means of non-repudiation reduces distrust since each party can obtain a proof of communication.

Consider an example, where a computer C_A of Bank A communicates with computer C_B of Bank B. C_A and C_B have a mutually authenticated channel for conducting financial transactions. Each bank requires each computer in its network to use a non-repudiation mechanism: only transactions with non-repudiatable proofs are accepted, and all proofs are stored. Now, suppose that C_A has malware controlled by an attacker who can use it to alter transaction amounts or transfer money to other accounts. Such malicious transactions will follow a path on Bank B’s network to a database. When a malicious transaction is detected, stored proofs can be used to trace the transaction to its origin (C_A) from its detection point.

Content Delivery Networks (CDNs). A CDN serves content on behalf of the owner of that content. The content can be a file [9]

¹An alternative approach is to include Online Certificate Status Protocol (OCSP) responses for the proof generator’s certificate at the time of evidence generation.

or a website [3]. A non-repudiation method allows a CDN user (e.g., a website visitor) to prove messages exchanged with a CDN. Similarly, a CDN can create various proofs for sessions that took place between a user and itself (if mutual authentication is enabled). This reduces the required trust between clients, content providers (CDNs) and content owners.

If a CDN is malicious and does not provide evidence over content it delivers, a client can report this behavior to the content owner. Also, robustness of non-repudiation would allow clients to obtain as much evidence as possible on the conversation considering that some of these systems suffer from reliability issues due to the complexity of the software stack behind.

Document Submission. A document submission system providing NRR allows a submitter to later prove that the ostensibly submitted document was indeed received by the submission system. If the document is large, submitter or the submission server may fail before the submission completes. This results in a submitter having no proof of (partial) submission. Similar faults can occur during importing public feeds, archiving websites and discovering misbehavior in peer-to-peer systems. Non-repudiation coupled with robustness can prevent loss of evidence in these scenarios.

3.3 Design Goals & Challenges

Robustness. TLS sessions can fail or abort due to errors occurring because of increasing complexity of software on both client and server sides. Although non-repudiation allows parties to combat these issues by later showing proofs of such events, due to unpredictability of errors and failures, some loss of evidence is unavoidable. The optimal approach to evidence loss is to generate evidence for each interaction. However, this is not possible due to TLS performance requirements. Therefore, we need a fine-tunable robustness mechanism to generate evidence with a certain frequency, which ensures minimization of maximum possible evidence loss.

Reducing Unfairness. A non-repudiation protocol is *fair* if it generates NRO and NRR evidences destined for the receiver and originator for the same message, respectively, without allowing either party to have an advantage over the other [24, 26]. In a mutually-authenticated setting where both parties act as a generator, fairness in non-repudiation refers to each party sharing the evidence on the same (subset of the) conversation between parties. Although this is achievable with cryptographic protocols with or without trusted parties [24, 25, 31, 40, 48], prior relevant techniques are unsuitable for TLS due to trust requirements (e.g., trust in a third party) or performance penalties. Therefore, our design should not attempt to guarantee non-repudiation fairness but rather should focus on reducing the potential evidence loss and ensuring that there are no large discrepancies between the amount of evidence available to the involved parties in a symmetric setting. Ideally, the design should be flexible enough to inform the other party what acceptable evidence loss is for a session participant, e.g., in terms of number of messages or content length.

Protection from Malicious Non-repudiation Entities. The goal of a malicious entity in our setting is to produce faulty non-repudiation proofs, i.e., for a modified session transcript. This might include convincing the other party to sign the wrong evidence. A natural approach is for each party to check correctness of every piece of

information received from the other party. Clearly, this must be done efficiently since performance is critical in a TLS setting.

Selective Non-repudiation. Although current methods preserve privacy of parts of a conversation, in certain cases, parties might need to exclude certain parts. For example, in peer-to-peer messaging, selective non-repudiation is desirable, since parties may want some parts of a conversation to be off-the-record while in other cases, parties should be able to forward messages in an authenticated way. Selective non-repudiation implies a discrepancy between the evidence generator and its verifier, since the latter needs to know which parts are deemed to be off-the-record by the generator. Also, sharing this information needs to be done efficiently.

Performance. In the performance-sensitive TLS setting, any non-repudiation approach should impose minimal burden. This includes computation and communication overhead, as well as the amount of state that parties maintain as part of the session.

3.4 ROSEN Properties

Based on the above discussion, we now present desired security and robustness properties. We adopt security definitions of [37] and extend them to provide symmetric non-repudiation. We also introduce robustness definitions regarding loss of evidence in case a party leaves a session either maliciously or due to a fault, and discuss efficiency requirements. Desired properties P_0 through P_4 , R_0 and E_0 are as follows:

CES-Unforgeability P_0 and CES-Privacy P_1 : [41] define CES-Unforgeability and CES-Privacy requirements for content extraction signatures (CES) for documents.

- CES-Unforgeability P_0 : A signature on a malicious document MD cannot be generated, such that the signature is valid and MD is not a part of the original document, or MD is a sub-document of the original document, however, the owner did not intend for MD to be accessible.
- CES-Privacy P_1 : An attacker cannot extract a sub-document (from the original document) that was not intended to be accessible by the owner.

Since the definition of a document is generic, these properties are directly applicable to our setting where a document is a complete TLS session transcript between a client and a server.

For efficiency reasons, we allow privacy decisions to be made over individual TLS records and “chunks” which are non-overlapping same-length sub-parts of TLS records, with the possible exception of the last chunk of a record being smaller in length than others.

Structure Property P_2 : The structure of hidden data is revealed by the proof. Hidden chunks or records in the proof (if any), must be distinguishable from complete chunks or records, respectively. Similarly, if there are redacted chunks, they must be explicitly disclosed in the proof.

Originator Property P_3 : The originator of each message is known, if it is intended by the proof generator.

Time Property P_4 : Tight bounds on generator’s time are provided in the evidence, for each checkpoint.

Robustness Property R_0 : Robustness Property R_0 requires minimizing evidence loss in case of a failed session.

Efficiency Property E_0 : Efficiency Property E_0 requires minimizing the state kept between checkpoints.

3.5 Definitions

We use the terms *Client (C)* and *Server (S)* to refer to TLS session participants or endpoints. The former initiates the TLS session. In addition, there are three roles in the targeted TLS setting:

- (1) *Generator*: TLS server and/or client (if client authentication is used). A generator has a public/private key-pair and signs evidence, i.e., messages it sends and receives. Signed evidence is returned to the requester upon demand.
- (2) *Requester*: a party that interacts with the generator in a TLS session. It can request evidence from the generator.
- (3) *Verifier*: a party that verifies evidence, i.e., non-repudiation proofs (NRO, NRR, NRC), produced by the generator. A verifier can be the requester or an external entity.

Note that, if a checkpoint mechanism is agreed upon, explicit evidence request messages are not used. Instead, the checkpoint schedule governs the periodicity of generating and sending evidence to the other party. In this scenario, we refer to the evidence receiving party as the requester and generating party as the generator.

Depending on the exact ROSEN threat model, we also describe a complementary *online entity*, an *evidence verifier*. This definition is due to the threat model where generators can try to deceive other parties by generating wrong evidence.

- (4) *Evidence Verifier*: Responsible for ensuring the correctness of the received evidence at each checkpoint, can be client or server.

4 PROPOSED TECHNIQUE: ROSEN

ROSEN improves on prior non-repudiation mechanisms by providing robust symmetric non-repudiation with additional features. ROSEN is based on periodically shared signed evidence, as depicted in Figure 3 and uses TLS-N [37] for evidence generation and requester-side redactions.

Implementation of checkpoints allows robust non-repudiation while allowing efficient proofs that use only a subset of the conversation. Furthermore, ROSEN provides *selective non-repudiation* which allows selective redaction of messages during evidence generation, allowing a generator to cull records (or parts thereof) for which it does not wish to provide non-repudiation. We also introduce the evidence verifier role to ensure correctness of evidence at checkpoints – i.e., the evidence is indeed over the conversation between them, not over an altered version of it.

In ROSEN, parties negotiate non-repudiation parameters during the initial TLS handshake. During the session, the generator keeps minimal state, while the requester stores messages between checkpoints. When the next checkpoint (as agreed in the initial handshake) is reached, the generator constructs and sends an evidence message to the requester. An evidence message includes a digital signature over the session state along with additional information. Each checkpoint only requires sharing minimal information. Evidence verifiers ensure the correctness of evidence.

We now describe each mechanism of ROSEN: parameter negotiation, checkpointing, followed by proof generation and verification.

4.1 Parameter Negotiation

A handshake is used to agree upon evidence and redaction granularity: *record-level* or *chunk-level*. With the former, commitments and redactions are on a per record basis, and per chunk basis for the latter. This is further detailed in Section 4.2. There is an optional parameter for checkpoint type:

- (1) **# records**: A checkpoint is triggered when the negotiated number of records sent by the party reaches this value.
- (2) **None**: *Request evidence* messages are used to determine when parties should return evidence. When to send these evidence request messages are decided by the sending party.

An alternative checkpoint agreement is possible, whereby the client sends a list of metrics it wants to use and the server picks one, similar to cipher suite selection in TLS. When a party reaches a checkpoint, or a request is received, evidence of all messages sent/received up to now is generated and sent to the other party. In addition, the handshake defines the roles (server *S* and client *C*), which are included in evidence generation. Number of records is only one option for a checkpoint metric. We discuss alternative options in Section 7.

4.2 TLS State

To provide symmetric non-repudiation, ROSEN relies on signed evidence messages shared at every checkpoint. To efficiently implement this exchange and message processing, we use a state maintenance scheme from TLS-N [37]. This scheme keeps minimal state (a hash) that includes incoming and outgoing messages, in the order observed by the generator. The scheme is based on hash chains whereby hash of each message in a conversation is chained to produce a final hash which is signed to create the evidence. Each message hash includes originator information: *S* or *C*. Messages hashes are generated with different methods depending on the granularity option which allows parties to redact content on a per record or per chunk basis. The redaction decision is made online whereas privacy of chunks can be preserved during proof generation. We now describe the methods used for each granularity.

Record-Level: hashes for each record are created by applying a cryptographic hash function to the record with a sufficiently-large salt to provide privacy. A unique salt is generated for each record by applying an expansion function $E()$ to the TLS traffic secret, combined with the record sequence number. This combination is expected to be unique per TLS 1.3 standard [11].

Chunk-Level: a Merkle Hash Tree (MHT) and a companion salt tree are used to provide chunk-level privacy and efficient generation/disclosure of salts. This privacy mechanism is the same as in TLS-N; refer to [37] for details.

Salt Tree Generation: We generate a unique salt for each chunk in a record using a salt tree. The TLS traffic secret concatenated with the record sequence number is used as the initial seed. An expansion function $E()$ is applied to this seed and the output is divided into (equal) left and right parts. $E()$ is recursively applied to each part, resulting in a binary tree where each leaf is a salt for

Figure 3: Protocol Overview

the chunk in the corresponding record index. This method only requires the disclosure of $\log c$ salts for a single hidden chunk.

MHT: Once salts are generated, hash based commitments are created using corresponding salts for each chunk. Then, by recursively hashing chunk commitments from the leaves to the root, a MHT is produced. This allows us to create efficient inclusion proofs for individual chunks. The MHT root is treated as the message of the corresponding record.

After generating message hashes for records, the originator bit (server or client), the message length and a context identifier are included for each message hash and the result is itself hashed. The context identifier is simply a bit to differentiate between hash chain nodes and message hashes.

4.3 Selective Non-Repudiation (SNR)

In many applications, the generator may not want to provide non-repudiation for the entire session. For example, in many secure messaging applications, deniability is an explicit goal [44]. ROSEN works in a hybrid setting where non-repudiation can be provided for parts of a session while other parts are deniable. We achieve this by providing *Selective Non-Repudiation (SNR)*, which, in contrast to TLS-N [37], does not only allow the requester to hide content of a session for confidentiality, but also lets the generator redact chunks of records and exclude selected data from evidence in order to maintain deniability.

The idea behind this is simple: Instead of including all record chunks in the MHT, for deniable chunks the generator replaces the commitments with a fixed constant bitstring (e.g., all zeroes) and informs the requester by including a bit-vector (similar to the ordering vector) called *redaction vector* that indicates which chunks were redacted; see Figure 4. Sending this redaction vector is necessary, because the requester cannot otherwise recompute the correct MHT root, other than by brute-force searching for the set of redacted chunks, which is infeasible.

Unfortunately, the size of the redaction vector is linear in the number of chunks per record, unlike the size of the origin vector, which only depends on the number of records. For a chunk size of 16B and a record size of 16kB, 1kB should be transferred *per record*

from the generator to the requester. Depending on the application, the generator may redact only a few contiguous chunks in one record (e.g., a sentence). Even though only one record includes a redacted chunk, the redaction vector would include all zeros for all other records.

To alleviate this problem, we implement a two-level redaction vector. At the first-level, denoted L_0 , a bit for each record reflects whether this record has at least one redacted chunk. At L_1 , only the redaction vector for chunks of the record is included. Both L_1 and L_0 redaction vectors are sent from the generator to the requester. If all messages include a redacted chunk, the additional number of bits (compared to the basic scheme of sending one bit per chunk) is only one bit per record. The redaction method is shown in Figure 4. Note that this can be improved, based on the specific application requirements. For example, if there are only a few chunks are redacted overall, only the indices for each redacted chunk can be sent. Section 7 outlines methods for further reducing the sizes of redaction vectors.

4.4 Checkpoints

We now describe the information included in checkpoints. Recall that, once parties agree on a checkpoint metric (as described in Section 4.1), TLS state is kept by generator(s) using the methods described in Section 4.2. If a party wants to redact records/chunks, redaction is applied using the mechanism in Section 4.3. With each incoming and outgoing record, the ordering vector is updated with originator information – a bit reflecting message origin ($1 - S, 0 - C$). If a generator redacts the message, L_0 and L_1 are updated to include redaction information. A checkpoint is triggered when the pre-agreed checkpoint condition is reached. In the first checkpoint, the evidence message is generated as follows:

- (1) The initial hash h_0 is computed over the following:
 - From the TLS Handshake:
 - C 's public key (if mutual-authentication is enabled)
 - S 's public key
 - Starting Timestamp
 - TLS Cipher Suite

Figure 4: Redaction Vector Optimization of ROSEN

- Salt Size
- Chunk Size
- Checkpoint option and metric for each party (if applicable, server then client)

From Evidence Generation

- Checkpoint timestamp
- The current TLS state (as described in Section 4.2)
- Redaction vectors L_0 and L_1

For the next checkpoints following values are used to calculate the checkpoint hash:

- Previous checkpoint timestamp
- Current timestamp
- The current ROSEN state h_i where $i \geq 0$
- Redaction vectors L_0 and L_1

Let us denote the final hash for both checkpoints h_{final} .

- (2) h_{final} is signed by the the generator and sent to the other party. Note that the checkpoints might occur at different times.

Note that there is constant processing of the TLS current state with new records. Therefore, each evidence contains the history of messages up to that point since the session start.

In addition to the signature on h_{final} , the generator sends to the requester the ordering and redaction vector.

Ordering Vector: Note that C and S might have different views of record order due to simultaneously sent messages. Therefore, the ordering vectors needs to be exchanged. This exchange is done at every checkpoint; it **only** includes the records between the last and the current checkpoints. Between checkpoints, the latest state of the checkpoint is preserved and the hash chain is advanced over this value.

Redaction Vector: Similar to the ordering vector, a redaction vector **only** stores redaction information of chunks after the last checkpoint, up to the next checkpoint.

To provide more symmetry, ROSEN can enforce that the number of received valid checkpoints differs by at most one for both parties at all times, which reduces unfairness as described in Section 3.3. The parties can ensure this by acting as an online evidence verifier and aborting the protocol if they receive invalid evidence or no evidence for a given checkpoint. In other words, a party should not sign and share a hash of the next checkpoint without getting the correct previous checkpoint from the other party. If a party decides to leave the conversation before reaching a checkpoint, it is advised to generate and send evidence. If this point in the conversation is before the first checkpoint, it is treated as the first checkpoint; if it

is after, it is treated as a later checkpoint in terms of information included in the evidence.

4.5 Evidence Verification

Non-repudiation evidence is only useful if it is generated over correct content. A malicious generator can try to cheat by changing conversation contents. Since prior TLS non-repudiation mechanisms do not offer a means to verify evidence *during the conversation*, any received evidence might be deemed invalid later, during proof generation.

To mitigate this behavior, a ROSEN requester verifies evidence at each checkpoint. To achieve this, the same state keeping mechanism discussed in Section 4.2 is instantiated on the requester side at the beginning of the TLS session.² The requester computes and stores hashes for each record. If selective non-repudiation is not enabled, the requester uses the ordering vector to re-order record hashes and generates its h_{final} over the conversation.

This hash is used (along with the generator's public key) to verify the generator's signature. If there is redaction, MHTs might need to be rebuilt. We discuss this mechanism and optimizations we make in Section 5.

4.6 Proof Generation/Verification

Non-repudiation of origin/receipt (NRO/NRR): an NRO/NRR proof includes: the message plaintext and its salt(s), public keys of the parties, along with other parameters and all hashes leading up to a checkpoint hash (in the hash chain) signed by either party.

Non-repudiation of conversation (NRC): An ordered list of records with their corresponding origins. Since TLS state is built upon ordered records, a signature on this state hash by either party provides NRC. Note that, NRC implies NRO and NRR.

5 IMPLEMENTATION & EVALUATION

We now describe ROSEN implementation and discuss evaluation results. We implemented ROSEN using Mozilla's Network Security Services (NSS) which consists of a set of libraries for both client- and server-side applications with security in mind. The libraries provide support for SSL, TLS and various versions of PKCS (Public Key Cryptography Standards), and certificates. Our implementation can be used by either a client or a server. ROSEN is a TLS 1.3 extension as defined in RFC 8446 [14], based on open-source TLS-N [12].

During the handshake phase, ROSEN extension is enabled alongside with client authentication. Extension messages are

²If symmetric non-repudiation is enabled – where both parties use ROSEN, this does not incur additional overhead.

used to agree upon a checkpoint mechanism, the checkpointing metric that determines when to exchange evidences, as well as TLS-N parameters: salt and chunk sizes. Salt size should be chosen to be big enough, such that brute-force attacks on records are infeasible. The current implementation uses 16-byte salts. We use the `ssl3_ExtAppendHandshakeNumber` function to specify handshake parameters on client or server and `ssl3_ExtConsumeHandshakeNumber` to retrieve these parameters from handshake messages.

For commitments, we use the SHA-256 which allows creation of “hiding” and “binding” commitments with sufficiently large salts. To expand these salts, we use HKDF – a key derivation function which does not reveal any information on its input. A combination of the TLS traffic secret and the nonce are used as input to HKDF. The TLS traffic secret is different for each party and the nonce is the record number. This combination is designed to prevent re-use of nonces.

Between checkpoints, only the state of the last checkpoint (a single hash) is kept and this state is updated with new messages using hash chaining. In addition, to minimize memory footprint of ROSEN, ordering and redaction vectors are deallocated after evidences are exchanged when a checkpoint is reached. They can also be stored for future proof generations. If redactions are not enabled, the party acting as verifier only needs to save the hashes of records between checkpoints. These hashes are then used to build the hash-chain according to the received order vector and verify the evidence. If redactions are possible, for verification, parties need to save records to rebuild the MHT using the updated (redacted) messages.

5.1 Latency Analysis

To evaluate ROSEN, we initialize a client and a server with the ROSEN extension enabled on a machine with following specs: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz chip with 16GB of RAM. Both parties use the state keeping mechanism (see Section 4.2) and process each incoming and outgoing message. In our experiment setting, *S* sends messages and generates evidence, while *C* checks correctness of the evidence it receives, i.e., acts as an evidence verifier.

Figure 5 shows record processing times for varying chunk and record sizes with no redaction on the server side. Record processing includes salt tree and MHT generation, followed by record hash calculation using the salt tree. For record-level granularity, record processing times remain constant for varying record sizes since no salt trees or MHTs are generated. For chunk-level granularity, they linearly increase with the size of the record since more chunks imply more nodes for the salt trees and MHTs. The results are averaged over 100 records for each chunk- and record-size combination.

To evaluate evidence generation and verification costs, we set the checkpoint trigger as one sent message, i.e., each sent message on the server side triggers evidence generation and client attests to evidence correctness after receiving the evidence. Since no redactions occur on the server side, evidence generation and verification do not depend on record or chunk sizes and take constant time: $\approx 4\text{ms}$ and $\approx 0.2\text{ms}$, respectively. Since the client keeps hashes for each record, evidence verification is fast. Figures 7 and 8 shows evidence generation and verification times on server and client,

Figure 5: Record processing latencies for record-level and chunk-level processing with different chunk sizes. No redaction on the server side.

respectively. Evidence generation includes creating an evidence message including ordering and redaction vectors and signed MHT root which attests to the conversation. Necessary information (e.g., public keys, as described in Section 4.4) is also included by using `PK11_DigestOp` function.

Figure 6 shows server latency overhead. M and R are checkpoint intervals and redaction rates, respectively. As the checkpoint interval, we use the number of records sent by the generator. Evidence generation takes constant time over different checkpoints. However, the sparsity of checkpoints changes loss of evidence in case a party departs the conversation.

Figure 9 shows evidence verification latencies depending on the redaction percentage on the server end. In our experiments, we first apply the redaction percentage to decide whether the record will have any redacted chunks and then the same percentage is used to decide whether a chunk should be redacted. The goal is to test the two-level redaction vector mechanism (from Section 4.3) and its effect on client latency.

If there are no redactions on messages, verification latencies are minimal. To achieve this, the client caches MHT roots for each message and reuses them in the hash chain after reordering. If there are redactions for selective non-repudiation, MHT roots need to be recomputed. Depending on the distribution of redacted chunks, this computation could be sped up by caching all MHT nodes; though, in the worst case, it would require almost the same amount of computation as without caching. We therefore opted to only caching MHT roots in order to reduce the memory footprint and still obtain performance improvements in the common case, when no chunks are redacted.

Even if there are redactions, salt trees remain the same for records. Therefore, the verification step can be sped up by caching the salt trees computed during record processing. Figure 10 shows client latencies when salt trees are cached. This optimization reduces the client latency by half. Evaluation results are for 4KB-each

Figure 6: Server latencies showing record processing and evidence generation times. A checkpoint is triggered when the number of sent messages on the server side is a multiple of 5 (a) and 10 (b). Messages are each 4KB and 16B chunk size is used.

Figure 7: Evidence generation latencies for record-level and chunk-level processing.

Figure 8: Evidence verification latencies for record-level and chunk-level processing. No redaction on the server side.

messages with 16B chunks. Similar to records, salt trees can be discarded after verification.

5.2 Storage Requirements & Memory Usage

To generate privacy-preserving non-repudiation proofs offline, clients (or servers if mutual authentication is enabled), at minimum, need to store (1) the TLS conversation, (2) the traffic secret used to generate the salts, (3) ordering and redaction vectors which will be used to re-order and redact records when needed, and (4) the signature(s) over the conversation. Using selective salt disclosure, efficient non-repudiation proofs then can be generated and shared with third parties. It is also possible to store e.g., salt trees to reduce offline computation costs but this is likely to incur unnecessary burden during the session.

During a TLS session with ROSEN enabled, parties keep a small state, a hash, over the conversation, at bare minimum. For generators, between checkpoints, ordering and redaction vectors must be kept and shared when a checkpoint is reached. Then they can be discarded. Ordering vectors require 1-bit memory usage per record whereas redaction vectors require 1-bit per chunk. With the optimization discussed in Section 4.3 however, this requirement can be minimized. We also discuss additional optimization techniques in Section 7.

Evidence verification requires parties to at least keep a list of hashes for each record after the last evidence they verified. These hashes can be discarded after evidence verification in the next checkpoint. However, the hash computed at the last checkpoint needs to be kept to advance the hash chain with the record hashes at the next checkpoint using the ordering vector. If selective non-repudiation is enabled, records after last evidence verification needs to be kept in order to redact and recalculate record hashes. MHT nodes might be kept to improve performance but this is not necessarily the case if corresponding chunks are redacted. Salt trees can be kept for records after the last evidence verification since

they are not affected by redactions. We show how this optimization improves performance in Section 5.1. Memory usage for a salt tree for a record is the number of chunks agreed upon during the handshake multiplied by the salt size.

5.3 Security Sketch

In this section, we give a security sketch as well as a robustness analysis of ROSEN. Our security sketch relies on the trust assumptions outlined in Section 3. A formal security analysis of TLS-N primitives underlying ROSEN are given in [37]. In this section, we briefly show that ROSEN modifications do not violate any of the TLS-N properties $[P_{0-4}]$. Then, we show how ROSEN satisfies remaining robustness and efficiency properties.

For CES-Unforgeability $[P_0]$ and CES-Privacy $[P_1]$, the proof directly follows TLS-N's proof since our redaction mechanism only overrides redacted chunks with zeroes and treat them as regular TLS-N chunks. For a more detailed security analysis of TLS-N for CES-Unforgeability $[P_0]$ and CES-Privacy $[P_1]$, we refer the reader to the TLS-N paper. Below we sketch how ROSEN does not violate any of these properties.

The unforgeability requirement is satisfied by the unforgeability of the signature scheme in conjunction with the collision resistance of the hash function. The privacy requirement is satisfied since hash functions are "hiding" when used with sufficiently large salts. However, these salts need to be pseudorandom and independent to provide privacy. KDFs can fulfill these requirements. Similarly, KDFs also provide input privacy where the traffic secret and the record sequence number are not leaked in the output.

Inclusion of redaction vectors in checkpoints ensures that they are disclosed as redacted messages. This process ensures that Structure Property $[P_2]$ is satisfied. Note that this does not violate the privacy requirement since the privacy of redacted messages are preserved by overriding the redacted chunks with zeroes.

For each record, ROSEN uses the bit value 1 to denote server messages whereas client messages are marked with the bit value 0. Since the originator information is included in the record hashes, Originator Property $[P_3]$ is satisfied when combined with CES-Unforgeability $[P_0]$.

To satisfy Time Property $[P_4]$, ROSEN includes time information at each checkpoint which in result provides time-bounded evidence generation. In fact, the time-bounds are tighter for ROSEN since each checkpoint include such information. One can consider time-based attacks where the attacker might include a wrong time in the evidence. However, such attacks can be detected by evidence verifiers.

In addition, ROSEN ensures that evidence losses are minimized by requiring receiving the evidence of the previous checkpoint until sharing the evidence on the next checkpoint, i.e., ROSEN satisfies Robustness Property $[R_0]$. This improves upon TLS-N's request-response based evidence sharing mechanism which is susceptible to evidence loss in case a party drops out of a conversation on purpose or unintendedly.

To adhere to property Efficiency Property $[E_0]$, ROSEN only keeps track of the last checkpoint state and ordering and redaction vector of only the chunks/records between checkpoints. After sharing the evidence when a checkpoint is reached, this information is

discarded by the evidence generator. Evidence verifier can only keep the records between checkpoints in case selective non-repudiation is enabled. If not, storing record hashes and using them for evidence verification is sufficient.

6 FURTHER RELATED WORK

ROSEN aims to provide a robust symmetric non-repudiation solution (especially) for systems requiring high assurance for provably tracking any mishaps during interactions of these systems. In addition, ROSEN reduces loss of non-repudiable data in case of a fault of entities occur, either maliciously or benignly. Lastly, it allows redaction of messages and generation of privacy preserving proofs. Therefore, the related work can be divided into three categories: in non-repudiation, privacy-preserving systems and robustness.

In Section 3 we discussed available solutions for providing non-repudiation and their drawbacks. Thus, in this section we give an overview of orthogonal solutions.

TLSNotary. TLSNotary [13] allows an auditee to prove to a third party that a conversation took place between a server and the auditee. It leverages the handshake mechanism of TLS 1.0 and TLS 1.1 to simulate non-repudiation where the auditor keeps some data secret that prevents the auditee from forging responses from server. This is achieved by abstaining the capability of generating server Message Authentication Code (MAC) keys of client by running a custom protocol between the auditee and the auditor.

After the auditee commits to the data obtained from the server, the auditor can safely share the secret data with the client. Since server responses cannot be forged by the client before this secret is shared, the auditor is convinced that committed conversation occurred between client and server. Unfortunately, TLSNotary cannot be used with TLS version ≥ 1.2 due to change in handshake mechanism. Another drawback of this mechanism is that it relies on the use of insecure [42, 43] hash functions SHA1 [18] and MD5 [38].

DECO. DECO [46], similar to TLSNotary [13], modifies the handshake mechanism of TLS to provide non-repudiation. Using the custom TLS 1.2 handshake mechanism, the client is divided into two parts as the prover and verifier.

At the end of the handshake phase, MAC keys are secret shared between the prover and verifier. For subsequent messages, a custom secure two-party computation is run to calculate and generate MAC tags between these parties. Zero-knowledge proofs are used to prove statements over the exchanged messages between the client and server. Unfortunately, due to the changes in the handshake mechanism of TLS 1.3, DECO requires modifications to be used within TLS 1.3. In addition, proposed custom three-party handshake mechanism allows creation of proofs that are acceptable *only* by the verifier, which needs to participate in the conversation between the client and the server from the beginning.

Town Crier. Town Crier [45] leverages Intel's Trusted Execution Environment technology Software Guard Extensions (Intel SGX) [7] to attest the code running on a CPU with SGX capability. The data obtained by this code (using a secure channel such as TLS) is then to be guaranteed to be originating from a trusted source (e.g., an authenticated server). The main purpose of Town Crier is to be able to import data to blockchain applications. Hardware

requirements reduces the deployability of such solution for the common Internet usage scenario.

Content Extraction Signatures [41] enable a party to extract verifiable content while enabling hiding some parts of the content. The authors propose schemes for allowing CEs and compare them in terms of computation, signature length and security. Moreover, they define requirements for CEs which are also requirements of ROSEN (See Section 3.4). Similarly, sanitizable signatures [15] allow censoring of messages (without interaction with the original signer) while allowing signature verification by a third party. Our message redaction can act to have *weak transparency* or *strong transparency*, as defined by the authors, depending on proof generation. In other words, a proof may choose to reveal that a message has been redacted (*weak transparency*), or not reveal it (*strong transparency*).

Robustness of a system defines how resilient a system is to faults. A fault, in general terms, is a misbehavior of an entity and fault tolerance is the ability of a system to remain in operation even in case of faults. Fault tolerance have been extensively studied in the literature ([17, 22, 23, 28, 36], to name a few).

In the context of non-repudiation a fault is behavior diverging from expected sharing of non-repudiation evidence inadvertently or deliberately. Similarly, for TLS sessions and non-repudiation, a fault can be benign, where a party drops the connection, therefore stops sending evidence on the conversation; or malicious, where a party aims to trick another party to sign wrong evidence or tries to deceive the other party by convincing that the evidence it generated is valid. In either case, the end result is that a party receives non-repudiation evidence only on a part of the conversation. [47] initially proposed a fair non-repudiation protocol and subsequent work [24, 25, 31, 40, 48] have made various improvements (Also See a survey on the issue [26]).

Unfortunately, fairness providing constructions either require a trusted third party or expensive cryptographic operations. Since our desire for TLS is to avoid both, we do not divulge into discussing such solutions. Rather, we focus on reducing evidence loss in case of faults by allowing participants to agree on an acceptable non-repudiation evidence loss for each party during the handshake.

Checkpoints to provide robustness have been proposed in the context of high-performance computing [34, 35, 39] and mobile systems [21]. Moreover, optimal checkpointing strategies for task completion was explored by [19]. We are not aware of any existing checkpoint mechanisms for providing robust non-repudiation.

7 DISCUSSION

In this section, we discuss possible extensions to ROSEN's checkpoint mechanism, metrics, application signaling for redaction and further optimizations.

In ROSEN, we design an efficient evidence verification mechanism based on verifying each evidence received. Instead, a randomized evidence verification can be employed by either party, i.e., at each checkpoint, client and server randomly choose whether to verify the evidence they received. Similar methods have been used before in the context of fairness in non-repudiation [31]. This method allows parties reduce computation costs by reducing the number of signatures verified – with the cost of possibly enabling attackers

avoid detection (for a period of time) for incorrect non-repudiation evidence.

On the other hand, if either party is expected to be benign, parties can choose to only store the evidence messages but not verify them during conversation. Such methods can be used, e.g., for CDNs, where clients collect non-repudiation evidence for future use. Some clients may choose to verify evidence during this period. A reputation system could be introduced to inform other clients when a server created wrong evidence. These solutions are, in essence, similar to gossip protocols for append-only logs with monitors [29, 32].

Our focus as a checkpoint metric for ROSEN has been number of messages sent by a party. This is certainly not the only feasible checkpoint metric. Depending on conversation properties, number of exchanged bytes, number of sent and received messages, even time from last checkpoint can be used by parties.

Since the redaction mechanism we implement lives in the transport layer, for applications that might need to signal the transport layer on which chunks to redact, one option is to use specific HTML tags e.g., `<redact></redact>`. When an application prefers to redact contents form the conversation, it marks them by placing these tags around them. ROSEN can parse these commands and redact accordingly. ROSEN can then redact records/chunks in-between these tags. This can allow ROSEN to be flexible regarding applications' choices of what parts of the conversation to redact. Furthermore, if the redaction decision is made by the application layer and both parties are aware of such a mechanism, this removes the need for the redaction vector.

Depending on the redaction frequency and nature of redactions, the size of redaction vector can be reduced to minimum by combining various approaches at checkpoints. For example, if number of redactions exceed the number of non-redacted chunks in records, at the corresponding checkpoint a party can inform the other party about how the redaction vector is structured for the current checkpoint. This allows flexible redaction vector encoding and in result reduces size of redaction vector. Similarly, the TLS handshake phase can be used to agree on a redaction vector reduction mechanism where the client sends a list of options and the server selects one. This mechanism is similar to cipher suite selection.

8 CONCLUSION

In this paper, we propose ROSEN – a robust non-repudiation mechanism for TLS that is suitable for mutual authentication (as well as one-way authentication) scenarios. ROSEN aims to reduce evidence loss by benefiting from a checkpoint mechanism that is agreed upon by each party. This checkpoint mechanism keeps minimal state by storing only needed information to verify received evidences.

While designing ROSEN, we consider different threat models including malicious non-repudiation entities and reliability issues each party might have. We evaluate ROSEN in a client-server setting by varying communication parameters (e.g., checkpoint metrics, record sizes). Results show that ROSEN puts minimal burden on parties and is flexible enough to be used in scenarios with different threat models. We hope that ROSEN will reduce distrust between entities by providing a robust mechanism for attesting to the communication between parties.

Figure 9: Client evidence verification and record processing latencies for different redaction percentages. Messages are each 4KB and 16B chunk size is used. A checkpoint is triggered when the number of sent messages on the server side is a 10.

Figure 10: Client evidence verification and record processing latencies for different redaction percentages. Salt trees are reused.

REFERENCES

- [1] [n. d.]. Authentication Method for Visa Payment Processing. <https://developer.visa.com/capabilities/vpp/docs-authentication>. ([n. d.]).
- [2] [n. d.]. Building Facebook's service encryption infrastructure. <https://engineering.fb.com/security/service-encryption/>. ([n. d.]).
- [3] [n. d.]. Cloudflare CDN. <https://www.cloudflare.com/cdn/>. ([n. d.]). Accessed: 2019-11-12.
- [4] [n. d.]. draft-hajjeh-tls-sign-04 - TLS Sign. <https://tools.ietf.org/html/draft-hajjeh-tls-sign-04>. ([n. d.]). Accessed: 2018-12-17.
- [5] [n. d.]. draft-housley-evidence-extns-01 - Transport Layer Security (TLS) Evidence Extensions. <https://tools.ietf.org/html/draft-housley-evidence-extns-01>. ([n. d.]). Accessed: 2018-12-15.
- [6] [n. d.]. Encryption in Transit in Google Cloud. <https://cloud.google.com/security/encryption-in-transit/>. ([n. d.]).
- [7] [n. d.]. Intel Software Guard Extensions. <https://software.intel.com/en-us/sgx>. ([n. d.]).
- [8] [n. d.]. Mutual TLS authentication. <https://developers.cloudflare.com/access/service-auth/mtls/>. ([n. d.]).
- [9] [n. d.]. Netflix | Open Connect. <https://openconnect.netflix.com/en/>. ([n. d.]). Accessed: 2019-11-12.
- [10] [n. d.]. Network Security Services. <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS>. ([n. d.]).
- [11] [n. d.]. RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3. <https://tools.ietf.org/html/rfc8446>. ([n. d.]). Accessed: 2018-12-16.
- [12] [n. d.]. TLS-N implementation for NSS. <https://github.com/tls-n/nss-tlsn>. ([n. d.]).
- [13] [n. d.]. TLSNotary - prove an https page was in your browser. <https://tlsnotary.org/TLSNotary.pdf>. ([n. d.]).
- [14] [n. d.]. The Transport Layer Security (TLS) Protocol Version 1.3. <https://tools.ietf.org/html/rfc8446>. ([n. d.]). Accessed: 2019-10-7.
- [15] Giuseppe Ateniese, Daniel H Chou, Breno De Medeiros, and Gene Tsudik. 2005. Sanitizable signatures. In *European Symposium on Research in Computer Security*. Springer, 159–177.
- [16] Gilles Brassard, David Chaum, and Claude Crépeau. 1988. Minimum disclosure proofs of knowledge. *Journal of computer and system sciences* 37, 2 (1988), 156–189.
- [17] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.
- [18] Donald Eastlake and Paul Jones. 2001. US secure hash algorithm 1 (SHA1). (2001).
- [19] Robert Geist, Robert Reynolds, and James Westall. 1988. Selection of a checkpoint interval in a critical-task environment. *IEEE Transactions on Reliability* 37, 4 (1988), 395–400.
- [20] Google. 2019. Transparency Report - HTTPS encryption on the web. (2019). Available from: <https://transparencyreport.google.com/https/overview>.
- [21] Hiroaki Higaki and Makoto Takizawa. 1998. Checkpoint-recovery protocol for reliable mobile systems. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*. IEEE, 93–99.
- [22] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6 (1984), 518–528.
- [23] Rolf Isermann. 2006. *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Science & Business Media.
- [24] Wojciech Jamroga, Sjouke Mauw, and Matthijs Melissen. 2011. Fairness in non-repudiation protocols. In *International Workshop on Security and Trust Management*. Springer, 122–139.
- [25] Kwangjo Kim, Sangjoon Park, and Joonsang Baek. 1999. Improving fairness and privacy of Zhou-Gollmann's fair non-repudiation protocol. In *Proceedings of the 1999 ICPP Workshops on Collaboration and Mobile Computing (CMC'99). Group Communications (IWGC). Internet'99 (IWI'99). Industrial Applications on Network Computing (INDAP). Multime*. IEEE, 140–145.
- [26] Steve Kremer, Olivier Markowitch, and Jianying Zhou. 2002. An intensive survey of fair non-repudiation protocols. *Computer communications* 25, 17 (2002), 1606–1621.
- [27] Leslie Lamport. 1981. Password authentication with insecure communication. *Commun. ACM* 24, 11 (1981), 770–772.
- [28] Jean-Claude Laprie. 1985. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15* (1985), 2–11.
- [29] Ben Laurie, Adam Langley, and Emilia Kasper. 2014. Certificate Transparency. *ACM Queue* 12, 8 (2014), 10–19.
- [30] Sergio Lerner. 2015. Renegotiation and TLSNotary. IETF TLS mailing list. (2015). Available from: <https://mailarchive.ietf.org/arch/msg/tls/8ZK37bSWPvHf0p0X9nFzVAoT-04>.
- [31] Olivier Markowitch and Yves Roggeman. 1999. Probabilistic non-repudiation without trusted third party. In *Second Conference on Security in Communication Networks*, Vol. 99. 25–36.
- [32] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. 2015. {CONIKS}: Bringing Key Transparency to End Users. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 383–398.
- [33] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.
- [34] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. 2010. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.
- [35] Nichamon Naksinehaboon, Yudan Liu, Chokchai Leangsuksun, Raja Nassar, Michaela Paun, and Stephen L Scott. 2008. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE, 783–788.
- [36] Brian Randell. 1975. System structure for software fault tolerance. *Ieee transactions on software engineering* 2 (1975), 220–232.
- [37] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, et al. 2018. TLS-N: Non-repudiation over TLS Enabling Ubiquitous Content Signing. In *Network and Distributed System Security Symposium (NDSS)*.
- [38] Ronald Rivest. 1992. The MD5 message-digest algorithm. (1992).
- [39] Sriram Sankaran, Jeffrey M Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. 2005. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *The International Journal of High Performance Computing Applications* 19, 4 (2005), 479–493.
- [40] Steve Schneider. 1998. Formal analysis of a non-repudiation protocol. In *Computer Security Foundations Workshop, 1998. Proceedings. 11th IEEE*. IEEE, 54–65.
- [41] Ron Steinfeld, Laurence Bull, and Yuliang Zheng. 2001. Content extraction signatures. In *International Conference on Information Security and Cryptology*. Springer, 285–304.
- [42] Marc Stevens. 2006. Fast Collision Attack on MD5. *IACR Cryptology ePrint Archive* 2006 (2006), 104.
- [43] Marc Stevens, E Bursztein, P Karpman, A Albertini, Y Markov, A Petit Bianco, and C Baise. 2017. Announcing the first SHA1 collision. *Google Security Blog* (2017).
- [44] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. 2015. SoK: secure messaging. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 232–249.
- [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*. ACM, 270–282.
- [46] Fan Zhang, Sai Krishna Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. 2019. DECO: Liberating Web Data Using Decentralized Oracles for TLS. *arXiv preprint arXiv:1909.00938* (2019).
- [47] Jianying Zhou and Dieter Gollman. 1996. A fair non-repudiation protocol. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*. IEEE, 55–61.
- [48] Jianying Zhou and Dieter Gollmann. 1997. An efficient non-repudiation protocol. In *Computer Security Foundations Workshop, 1997. Proceedings., 10Th*. IEEE, 126–132.