maTLS: How to Make TLS middlebox-aware?

Hyunwoo Lee[†], Zach Smith[§], Junghwan Lim[†], Gyeongjae Choi[†],
Selin Chun[†], Taejoong Chung[‡], Ted "Taekyoung" Kwon[†]

[†]Seoul National University, [§]University of Luxembourg, [‡]Rochester Institute of Technology
[†]{bir1218,twoexponents,ryanking13,littlechun4,tkkwon}@snu.ac.kr, [§]zach.smith@uni.lu, [‡]tjc@cs.rit.edu

***Abstract*—Middleboxes are widely deployed in order to enhance security and performance in networking. As communication over TLS becomes increasingly common, however, the end-to-end channel model of TLS undermines the efficacy of middleboxes. Existing solutions, such as 'SplitTLS', which intercepts TLS sessions, often introduce significant security risks by installing a custom root certificate or sharing a private key. Many studies have confirmed security vulnerabilities when combining TLS with middleboxes, which include certificate validation failures, use of obsolete ciphersuites, and unwanted content modification. To address the above issues, we introduce a middlebox-aware TLS protocol, dubbed maTLS, which allows middleboxes to participate in the TLS session in a visible and auditable fashion. Every participating middlebox now splits a session into two segments with their own security parameters in collaboration with the two endpoints. The maTLS protocol is designed to authenticate the middleboxes to verify the security parameters of segments, and to audit the middleboxes' write operations. Thus, security of the session is ensured. We prove the security model of maTLS by using Tamarin, a state-of-the-art security verification tool. We also carry out testbed-based experiments to show that maTLS achieves the above security goals with marginal overhead.**

I. Introduction

Middleboxes have been widely used for various in-network functionalities and have become indispensable. They are usually deployed by network operators, administrators, or users for various benefits in terms of performance (e.g., proxies, DNS interception boxes, transcoders), security enhancement (e.g., firewalls, anti-virus software), or content filtering (e.g., parental controls). Such deployments have become easier and more flexible with the advent of cloud computing represented by 'everything-as-a-service,' including outsourced middleboxes as a service in the cloud [40].

However, the practice of using middleboxes is not compatible with Transport Layer Security (TLS) [9], [12] — the de-facto standard for securing end-to-end connections. Since TLS is initially designed to provide *end-to-end* authentication and confidential communication, middleboxes are not supposed to read or modify any TLS traffic. Meanwhile, as HTTPS (HTTP over TLS) [37] becomes increasingly common (more than 50% of total HTTP traffic is now encrypted by TLS [14], [28]), middleboxes are at risk of becoming useless unless a solution is found. To address this issue, several approaches have been made to retain the function of middleboxes over HTTPS.

A well-known method is SplitTLS [19], in which a TLS session between two endpoints is split into two separate segments[1] so that a middlebox can decrypt, encrypt, and forward the traffic as a man-in-the-middle. While SplitTLS allows us to use middleboxes with TLS, it poses security and privacy risks on both the client and server sides. On the one hand, users are often required to install custom root certificates, which allows a middlebox to impersonate any server in order to read and modify all the HTTPS traffic. On the other hand, HTTPS websites often share their private keys with some middlebox service providers (e.g., content delivery networks (CDNs)), so that middleboxes can provide their content to clients with better performance. These imply that *a compromised middlebox may be used to perform critical attacks*, either by abusing custom root certificates to impersonate someone else or by using a shared private key to impersonate a particular server.

Such vulnerabilities of middleboxes have been reported in several studies [8], [11], [46], [34], [44]; for instance, some middleboxes accept *nearly all certificates* in spite of certificate validation failures, which gives a chance for another compromised/malicious middlebox to meddle in the TLS session [8], [11], [46]. Similarly, a middlebox that splits a TLS session may support only weak ciphersuites, which are vulnerable to known attacks such as the Logjam attack [1] or the FREAK attack [3]. Even worse, it has been reported that middleboxes are being used to inject malicious code [44], [34], [5]; for example, Giorgos et al. [44] found that 5.15% of proxies inject malicious or unwanted content into web pages.

Nevertheless, as middleboxes provide crucial benefits to users, content providers, and network operators, there has been a long thread of studies aiming to accommodate for middleboxes in secure networking between two endpoints [41], [20], [35], [16], [22], [31], [30]. These studies can be largely classified into three main categories: encryption-based, trusted execution environment (TEE)-

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23547
www.ndss-symposium.org

[1] In this paper, an end-to-end channel between a client and a server is called a TLS (or maTLS) session, while a channel between two points at which TLS messages are encrypted and decrypted with the same key, respectively, is called a TLS (or maTLS) segment.

based, and TLS extension-based. First, BlindBox [41] and Embark [20] proposed to use special encryption schemes such as order-preserving encryption to allow middleboxes to perform their functionality over encrypted packets. Second, SafeBricks [35] and SGX-Box [16] leveraged TEEs such as Intel SGX to make middleboxes trustworthy. Third, several studies sought to extend the TLS protocol [30], [22], [31], [23], [29] in order to let middleboxes intervene during the TLS handshake and perform their functionalities within the session.

However, these approaches pose several technical challenges and limitations. The encryption-based approaches depend greatly on their encryption mechanisms; as a result, their functionalities are limited to pattern-matching or range-filtering. The proposals leveraging TEEs are only applicable to the middleboxes with specific hardware that provides secure enclaves. What is worse, neither of them are backward-compatible (i.e., current middleboxes have to be replaced to adopt such approaches). The TLS extension approaches are most feasible in the sense that TLS software can be extended to support the backward compatibility. However, these approaches leave three issues that have not been comprehensively solved.

First, the proposal of using explicit proxies in IETF [22] introduces a proxy certificate to indicate that the certificate holder is a middlebox. However, the client can only authenticate the next middlebox, not the server or other middleboxes intervening in the session. Thus, there is still a risk of an unknown middlebox meddling in the session. Second, mcTLS [30], TLMSP[2], and TLS Keyshare extension[3] [31] use the same symmetric key (and hence the same ciphersuite) across all the split TLS segments between the two endpoints. As a result, middleboxes that do not support the specific ciphersuite chosen will not be able to process the TLS traffic. Furthermore, the middleboxes share the same keystream, which may undermine confidentiality [23]. Third, none of these proposals except TLMSP allow the client to know who has sent TLS traffic as well as who has modified it. For example, in mcTLS [30], the client cannot check whether the TLS traffic he received originated from a valid endpoint (e.g., a cache or an endpoint) if there is a middlebox that modified the message during transit.

In this paper, we propose an extension to TLS, which ensures middleboxes are *visible* and *auditable*. The starting point is to enable a client to authenticate all the middleboxes. We first define *middlebox certificates*, which are signed by certificate authorities (CAs), and used to encrypt the channel for each TLS segment (e.g., between a client and a middlebox, between middleboxes, and between a middlebox and a server). The use of middlebox certificates eliminates the insecure practice of users installing custom root certificates or servers sharing their private keys with third parties (like CDNs). We also introduce them with middlebox transparency log servers to make middleboxes auditable. Along with auditable middleboxes, we design the middlebox-aware TLS (maTLS) protocol, a TLS extension auditing the security behaviors of middleboxes. The maTLS protocol is designed to satisfy the following security goals (to be detailed later): server authentication, middlebox authentication, segment secrecy, individual secrecy, data source authentication, modification accountability, and path integrity.

To satisfy these goals, a client authenticates all participants of its maTLS session. That is, the client verifies the certificates of all the participating middleboxes to prevent any arbitrary middleboxes from intervening in the session, which we will refer to as *explicit authentication*. Moreover, the two endpoints confirm the negotiated security association of every segment to ensure its confidentiality and integrity, which is called *security parameter verification*. Note that a security association consists of a TLS version, a ciphersuite, and a confirmation of encryption key establishment. Lastly, maTLS performs *valid modification checks*, which allows the endpoints of a maTLS session to verify whether the received messages have been modified only by authorized middleboxes. This way, maTLS provides auditability of all participants in the session.

We also evaluate the security and performance of maTLS. We formally prove the security of maTLS with Tamarin [24], a state-of-the-art symbolic verification tool. We also implement maTLS by leveraging OpenSSL to compare its performance against prior proposals.

The remainder of the paper is organized as follows. First, we present the background of middleboxes and detail the problems with SplitTLS, while clarifying the security-related definitions and concepts (§II). Next, we explain our trust and threat model (§III). Then, we describe how to make middleboxes auditable (§IV), and design the maTLS protocol (§V). We verify our security model (§VI), evaluate the performance overhead of maTLS (§VII), and discuss further issues (§VIII). Finally, we summarize the related work (§IX) and present our concluding remarks (§X).

II. Background

A. Transport Layer Security

The TLS protocol [9], [12], coupled with a Public Key Infrastructure (PKI), is designed to authenticate endpoints, establishing a secure communication channel between them. The security goals of TLS are authentication, confidentiality, and integrity: *authentication* is confirmation of the identity of the other party, by validating a certificate chain and verifying a proof-of-possession of the corresponding private key. In practice, the server is always authenticated from its certificate, while authenticating the client is optional. *Confidentiality* is a guarantee that the data sent over the channel is secret to all but the endpoints. *Integrity* ensures that any third parties do not modify data on the network.

These security goals are achieved by two components of the TLS protocol suite, called the handshake and record

[2] Transport Layer Middlebox Security Protocol (https://portal.etsi.org/webapp/WorkProgram/Report_WorkItem.asp?WKI_ID=52930). The protocol is being discussed in ETSI, and the draft of the protocol specification is currently unavailable. We refer to the document in the web archive: https://docplayer.net/88122390-Announcement-of-middlebox-security-protocol-msp-draft-parts.html

[3] Note that this is different from the `keyshare` extension used to negotiate a Diffie-Hellman shared key in TLS 1.3.

protocols. The main purpose of the TLS handshake protocol is to establish a master secret, which will be used for an authenticated encryption and decryption of the data between two endpoints.

B. X.509 Certificates

A digital certificate is an attestation that binds a subject (e.g., a domain name) to its public key. This binding is guaranteed by a Certificate Authority (CA) with its signature in the certificate. The CA also possesses its certificate issued by another CA. This results in a chain of certificates terminated with a self-signed certificate called a root certificate. A certificate receiver validates the certificate if the receiver trusts the root certificate in the chain and all the signatures in the certificates can be verified using the public key of the next certificate in the chain (up to the root certificate).

CAs also indicate that a domain owner satisfies specific suggested requirements. For example, a domain validation (DV) certificate is issued when a domain owner has successfully proved its ownership of the domain. To provide stronger assurance to clients that a certificate has been adequately issued, CAs can require domain owners to follow a set of stricter criteria in order to obtain extended validation (EV) certificates.

On the Internet, X.509 [18] is the most widely used format for certificates, which typically include fields such as the subject, its public key, a serial number, and the certificate's validity period. The current version of X.509, version 3, supports extensions that CAs can add for a variety of purposes; for example, the Server Alternative Name (SAN) field [7] is used to allow alternative names of the certificate holder.

C. Certificate Transparency

The PKI trust model has a severe drawback in reality: any CA can issue a certificate for *any* domain, potentially exposing users to high risk. There have been security incidents in which commercial CAs were compromised and issued fraudulent certificates, allowing attackers to impersonate the actual certificate owner or perform man-in-the-middle attacks [6], [47].

To mitigate the risks from CA compromises, Google introduced the Certificate Transparency (CT) system [21], which aims to provide *accountability* to a PKI. This is achieved by archiving every certificate into multiple append-only public log servers so that any entity can monitor and audit a CAs' operations. Upon submission of a certificate chain, the log servers return a signed proof called a signed certificate timestamp (SCT), which can be verified using the public keys of the log servers. An SCT can be delivered from web servers to the browsers separately or embedded in the web server's certificate, via a TLS extension or through OCSP. For example, a browser might display a lower security indicator if the server's certificate is not logged on the CT servers. CT logging became mandatory in Chrome for all certificates issued after April 2018 [33]. A third party (e.g., a CA) can keep track of CT log servers to see if there is any mis-issuance of certificates, thus providing *auditability* of certificates and *accountability* of CAs' certificate issuance. For example, TLSMate's CertSpotter [42] and Facebook's CT Monitor [13] monitor each log server and alert a domain owner if a new certificate that binds to her domain name has been issued.

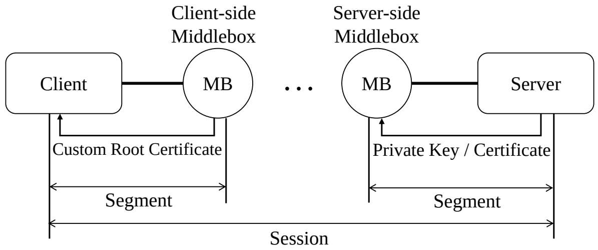


Fig. 1: **Overview of SplitTLS:** A client sets up a TLS session with a server involving multiple middleboxes in-between. During a TLS handshake, each middlebox splits the TLS session into two TLS segments while pretending to be the client's intended server. To this end, client-side middleboxes install custom root certificates on clients' devices and fabricate the server's certificate signed by the custom root certificates. On the other hand, server-side middleboxes take private keys from the server in order to impersonate the server in SplitTLS.

D. Middleboxes in SplitTLS

In this paper, we consider middleboxes which inspect application data sent over HTTPS, for the purpose of security or performance. Figure 1 illustrates how they typically intervene in a TLS session. A middlebox intercepts the TLS session, splitting it into two segments. The middlebox then pretends to be the client while communicating with the server and in turn impersonates the server in its communication with the client. In the case of multiple middleboxes, they form a chain of TLS segments between the client and server, with each middlebox ultimately playing both the roles of client and server during each round trip.

Once the end-to-end session is established, the client and the server communicate via the middleboxes. When a middlebox receives an encrypted message over a segment, it decrypts the message using the key of the segment. Then, the middlebox performs its functionality on the decrypted message. Finally, the middlebox encrypts the message with the key for the next segment and forwards it to the next middlebox (or the endpoint). Note that we are interested only in those middleboxes that participate in two *segments* simultaneously; for instance, we do not consider middleboxes that play the role of the intended servers to service the content such as edge servers in CDNs, since they do not always participate in two segments.

Depending on which entity installs the middleboxes and where they are deployed, we can classify middleboxes into two categories: client-side and server-side. **Client-side middleboxes** are employed by users (e.g., anti-virus software) or operators of client-side networks (e.g., intrusion detection systems). They are located at vantage points which packets always pass through. For example, a secure gateway, such as Bluecoat system[4], can be situated at the edge of a corporate network to inspect all the incoming and outgoing packets. **Server-side middleboxes** are deployed by web servers or by the contracts between the web servers and middlebox service providers. They are deployed on a server's networks, or in clouds that provide middlebox-as-a-service [40]. A client typically accesses server-side middleboxes through DNS routing. For example, when a server employs an outsourced web application firewall, such as Cloudbric[5], he changes the DNS zone file in his authoritative name server to direct traffic from clients to the firewall. After the firewall's inspection, the traffic is then forwarded to web servers or to further middleboxes based on the IP address configuration in the firewall settings.

Also, different techniques are used to intercept TLS sessions, depending on the middlebox type. For a client-side middlebox, clients are often required to install custom root certificates into the trusted root certificate store on their devices. Whenever a middlebox receives a TCP SYN packet sent to the server from the client, it intercepts the packet, executing a TCP handshake and then performing a TLS handshake with the client. During the TLS handshake, the middlebox generates a new certificate on-the-fly with the same common name as the intended server, which is signed by the private key that corresponds to the custom root certificate. Thus, if an attacker learns any private key of a custom root certificate, he can impersonate any server to which the client that trusts the custom root certificate wishes to connect. Furthermore, as the certificate is not issued by CAs, clients cannot verify its legitimacy by other means, such as through CT or DANE [38]. For server-side middleboxes, web servers are required to hand over their private keys along with the certificates so that the middleboxes can service their content. This breaks the fundamental principle of authentication and weakens the security of the servers, which makes middleboxes attractive targets for attackers [23], [4].

E. Security Problems in SplitTLS

Although SplitTLS complies with the current TLS practice, several studies have reported that some middleboxes fail to correctly validate certificates, degrade to weaker ciphersuites, or insert malicious scripts [8], [11], [44], [5]. This means that fundamental security properties (i.e., authentication, confidentiality, and integrity) between two endpoints are broken. The client is forced to trust the behavior of middleboxes, since the security of the session is highly dependent on whether the middleboxes correctly operate the TLS protocol. We summarize how SplitTLS breaks the security goals of TLS.

Authentication: A client cannot authenticate the intended server, as the middlebox replaces the server's certificate with a certificate forged by the middlebox. Even worse, recent studies showed that some middleboxes do not validate the certificate of the intended server. For example, PrivDog [2] was known to accept *every* certificate without checking its validity, and some anti-virus software *always* generates valid certificates even when it received invalid certificates from the intended servers (or another middlebox) [8], [5]. **Confidentiality:** Because a middlebox splits the original session into two segments, the client negotiates the key for the segment with the middlebox, not the intended server. Thus the middlebox can read or modify all traffic between the client and the server. Further, the client has no idea of whether the data has been encrypted (with a strong ciphersuite) after it passes through the middlebox. For example, when a client sends an HTTPS request to a server by using Nokia's Xpress Browser, it forcibly sends all messages to the Nokia's forward proxy. Then, this proxy delivers the messages on behalf of the client to the server. However, the Xpress Browser does not notify the clients that their information can be read or modified by the proxy [25], [15].

Integrity: SplitTLS cannot guarantee the integrity as a client cannot detect any modification by a middlebox on her messages with the intended server. For example, Lenovo laptops performed a man-in-the-middle attack to inject sponsored links on web pages (delivered over TLS) using Superfish [39], but this injection behavior was not noticeable by the ordinary client.

The above problems take place mainly because it is difficult for a client to detect which middleboxes meddle in the session and what they do to the traffic. Therefore, we propose that making middleboxes *visible* to clients and publicly *auditable* will help to address the above security and privacy challenges.

III. Trust and Threat Models

Entities. Before introducing our threat model, we describe five entities in the networking architecture.

(1) *Client (C)*: A client refers to a machine or a piece of software (e.g., web browsers), used by a *user*, that communicates with middleboxes. We assume the client correctly performs protocols and is not compromised.

(2) *Server (S)*: A server refers to a machine or a piece of software, operated by a *content provider*, that services content based on a client's request. We assume that the server to which a client wishes to connect is not malicious or compromised. The client and the server are collectively referred to as *endpoints*.

(3) *Middlebox (MB)*: a middlebox is a machine or a piece of software, made by a *middlebox service provider*. A middlebox is deployed by a network operator, a content provider, or a user and is located between the client and the server. The endpoints may not be aware of the middleboxes, their functions, or their states. If the middleboxes are mis-configured or incorrectly implemented, they may accept invalid certificates, use deprecated ciphersuites, or attempt to inject unwanted or malicious content [44], [34].

(4) *Certificate Authority (CA)*: An organization that issues and revokes certificates. A CA issues a certificate to

[4] https://www.symantec.com/products/proxy-sg-and-advanced-secure-gateway

[5] https://www.cloudbric.com/

a requester after a validation process. In our model, A CA can be compromised; thus, fraudulent certificates can be issued to an adversary who can impersonate the server.

(5) *Middlebox transparency (MT)*: A system (similar to CT [21]) that logs certificates, which can be publicly monitored and audited by any interested parties. Any *trusted* CT operator, such as Google, can operate an MT system. The only difference from CT is that the MT system targets *middlebox certificates*, which will be detailed in Section (§IV). Alternatively, the CT system can be assumed to accommodate middlebox certificates as well.

Adversary capabilities. We accept the Dolev-Yao model [10] in which an active adversary can fully control the network; that is, the network is untrusted. The adversary can not only capture messages on-the-fly, but also modify, drop, reorder, or inject messages. Specifically, he can manipulate middleboxes (e.g., TLS-intercepting WiFi access points), which then can capture packets, perform crypt-analysis, or patch software to inject malicious scripts. We do not consider other attacks such as side-channel attacks or denial-of-service attacks. .

IV. Auditable middleboxes

In this section, we describe an architecture to make middleboxes visible to the endpoints of TLS sessions. To this end, we define the notion of an *auditable middlebox* that has its own *middlebox certificate* logged in *middlebox transparency* (MT) servers. Middlebox certificates are written based on the X.509 format, and then signed by CAs, which may require middlebox service providers to follow a set of established criteria for certificate issuance. Like TLS certificates, middlebox certificates could also be mis-issued, mis-configured, or exploited. To mitigate those attacks, we also introduce *MT log servers* where any middlebox certificates can be publicly logged so that interested parties can monitor and detect unexpected behaviors.

A. Middlebox Certificates

The primary purpose of middlebox certificates is to help users authenticate middleboxes by providing the information about behaviors of the middlebox; for example, the role of the middleboxes (e.g., firewall) or permissions (e.g., read or write) can be included. This information can be added into the format of X.509 certificate without any modification to the existing infrastructure. Below, we itemize the required information for a middlebox certificate along with the names of the fields.

- **Name(s) of the Middlebox Service Provider** indicates the name(s) of the middlebox service provider, which can be specified at the `CommonName` field.
- **Subject (Middlebox) Public Key Info** carries the public key and the cryptographic algorithm (e.g., ECC) used to generate the key, which can be specified at the `Subject Public Key Info` field.
- **Middlebox Information Access** contains additional information that can help a user trust the middlebox. To this end, we define an extension, `Middlebox_InfoAccess` where its ASN.1 syntax is defined as follows.

```
Middlebox_InfoAccess :: =
  SEQUENCE SIZE (1..MAX) OF Middlebox_Description

Middlebox_Description::=  SEQUENCE {
  Middlebox_InfoType      OBJECT IDENTIFIER,
  Middlebox_Info          GeneralName}
```

  For example, *permission* can be one of the `Middlebox_InfoType` fields, used to indicate the read or write permission required by the middlebox for TLS traffic. Similarly, the *TypeofService* and *URL* fields can provide additional information about the middlebox as a form of `Middlebox_Description`.

B. Middlebox Transparency

We introduce an MT log server that publicly records middlebox certificates. The operation of MT is similar to that of CT [21]. It encourages middlebox service providers or CAs to submit middlebox certificates to the MT log server. Further, once a middlebox certificate is accepted at the MT log server, the log server returns a Signed Certificate Timestamp (SCT). A client can check its membership by verifying the SCT with the public key of the log server.

C. Properties of Auditable Middleboxes

We call a middlebox that has a middlebox certificate logged in an MT log server an *auditable middlebox*. It provides the following benefits regarding the *trustworthiness* of middleboxes:

First, middleboxes now have their own key pairs and can be authenticated from the endpoints by presenting their *valid* certificate. Thus, middleboxes now no longer require (1) content providers to share their private keys or (2) users to install their custom root certificate.

Second, clients can be assured of the names and properties of middleboxes or middlebox service providers. This will hold middlebox service providers accountable.Further, with the help of maTLS, which will be detailed in §V, clients can detect if a middlebox has modified traffic without any authorization. This can be done by checking the *Permission* item in the `Middlebox_InfoAccess` field of the middlebox certificate, which would encourage middleboxes to have *least* privileges. For example, anti-virus software can be issued with a middlebox certificate with only read permission to assure users that it will not modify any traffic.

Third, middlebox certificates may require some of the essential X.509 extensions such as *Permission* field to be set to `critical` [18], which explicitly indicates that clients must refuse the connection if they cannot interpret the extension.

Fourth, the MT system provides a global set of auditable middleboxes; any interested parties, such as monitors, auditors, and clients, can check any mis-issued, mis-configured, or fraudulent certificates.

Fifth, when a middlebox certificate's corresponding private key is no longer safe due to security breaches, the middlebox certificate can be revoked, and the revocation status can be disseminated through existing revocation mechanisms such as CRL [7] or OCSP [26]. Thus, clients can be protected from middleboxes with security risks by leveraging the existing revocation mechanisms.

Given that the PKI has been suffered from many security issues regarding certificate management, one might be concerned that introducing additional infrastructure (i.e., MT system) could exacerbate the current situation. However, we believe that the middlebox certificate by itself *does not* introduce new management problems as it can be easily integrated into the existing CT architecture. Rather, the use of middlebox certificates can mitigate the current insecure practices of middleboxes splitting TLS connections such as installing custom root certificates or sharing private keys.

V. Middlebox-aware TLS (maTLS)

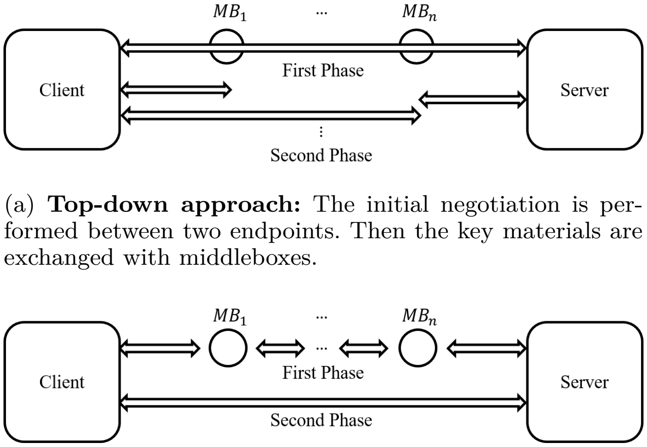
In this section, we describe the maTLS protocol, which is designed to allow middleboxes to participate in a TLS session. As we have middleboxes equipped with certificates, we extend the security goals of TLS to the seven objectives below, divided into three categories. For the sake of exposition, we explain maTLS based on TLS 1.2 with ephemeral Diffie-Hellman (DHE) key exchange in the server-only authentication mode.

A. Security Goals

Authentication: Similar to the authentication process of TLS certificates, clients should be able to receive and check the validity of the certificate of the server that the clients intended to connect. This should hold even when there are middleboxes splitting the TLS connection between them. Thus, we extend the notion of the authentication to cover both the intended server and middleboxes, and we call this property of the maTLS protocol (1) *Server Authentication.* Clients should also be able to authenticate the middleboxes by verifying the middlebox certificates, which we call (2) *Middlebox Authentication.*

Confidentiality: Browsers warn a user if her session is negotiated with a low TLS version or a weak ciphersuite. Thus, each maTLS segment should be encrypted with a sufficiently high version of TLS and a strong ciphersuite; we apply this requirement to each maTLS segments, which is called (3) *Segment Secrecy.* Further, each maTLS segment should have its own security association (e.g., a unique session key) to prevent the same keystream from being reused across the overall maTLS session. This goal is called (4) *Individual Secrecy.*

Integrity: The notion of integrity can be extended such that only authorized entities can generate or modify messages depending on their permissions. To this end, we define (5) *Data Source Authentication*, which means that a client should be able to confirm that a received message has originated from a valid endpoint such as a web server or cache proxy. Moreover, a client should be able to figure out which middleboxes have made each modification to the message, ensuring accountability. We call this (6) *Modification Accountability.* Moreover, not only the integrity of the messages should be preserved, but also the order of the middleboxes; the network attacker could also capture and redirect packets, or bypass some middleboxes. Therefore an endpoint should be able to confirm that all messages passed through the authorized middleboxes in the established order. We call this property (7) *Path Integrity.*



(a) **Top-down approach:** The initial negotiation is performed between two endpoints. Then the key materials are exchanged with middleboxes.

(b) **Bottom-up approach:** The two participants of each maTLS segment negotiate security parameters independently, and then the maTLS session is established by connecting the maTLS segments.

Fig. 2: Two approaches to establish a TLS session with middleboxes. We adopt the bottom-up approach since it efficiently supports incremental deployment.

B. maTLS Design Overview

Session Establishment Approaches: First of all, we explain how a client establishes a maTLS session with the server through multiple middleboxes. There are two possible approaches to establish a maTLS session and its segments, as shown in Figure 2. In the top-down approach, the client first establishes a TLS session directly with the server, and the server determines the security parameters of the session. After that, either or both of the endpoints should pass the segment keys to the authorized middleboxes via separate TLS connections. In the bottom-up approach, the client and middleboxes first initiate TLS segments sequentially up to the server. In this approach, the two participants of each segment negotiate their security parameters individually, and the session is eventually constructed from these segments.

In maTLS, we adopt the *bottom-up approach* for the following reasons. First, an maTLS session can be partially established even if not all entities support maTLS. For example, even if the server does not support maTLS, the client and the next middlebox that supports maTLS can still negotiate security parameters for their segment and establish a maTLS session. Second, each different maTLS segment can benefit from using strong ciphersuites or newer TLS version independently because maTLS does not require all entities to share the same ciphersuite or TLS

Audit Mechanism	Proof Data Structure	Description & Advantages
Explicit authentication	A sequence of certificate blocks, including the server certificate and any middlebox certificates with their signed certificate timestamps.	The client authenticates the server and middleboxes by checking their certificates, and confirms their names and the middleboxes' permissions • No custom root certificate and no private key sharing • EV certificates are not degraded due to fabricated certificates • Support for Certificate Transparency [21] and DANE [38]
Security parameter verification	Security parameters of every maTLS segment including a negotiated TLS version, an agreed ciphersuite, and a transcript of the handshake	The client confirms the confidentiality of every segment • Neither a low TLS version nor a weak ciphersuite is permitted without the client's knowledge • The two points of each segment perform a TLS handshake and establish a segment key
Valid modification checks	A modification log that keeps track of the modifications of a packet	The client confirms that only authorized entities can generate or modify messages • Only an authorized data origin (a server or a cache proxy) can generate messages • Only trusted writer middleboxes can modify messages • The order of middleboxes is always preserved

TABLE I: **Three audit mechanisms of endpoints in maTLS:** Explicit authentication guarantees the authentication of all the participants. Security parameter verification ensures the confidentiality of all the maTLS segments. Valid modification checks ensure that only authorized entities can modify messages.

version. Third, the bottom-up approach efficiently achieves *Individual Secrecy*. This is because the two entities involved in each segment use different random numbers to establish a master secret; thus, the probability that all the segment keys are identical is negligible.

It is worth noting that most of the top-down approach schemes, such as mcTLS [30], TLMSP, and TLS Keyshare extension [31], do not support incremental deployment. This is mainly because only the server picks the version, ciphersuite, and extensions that are supported across all entities (i.e., both endpoints as well as middleboxes), which makes it challenging to deploy them incrementally. Even worse, it is highly likely that the security level of the session will be decided by the "intersection" of the security parameters supported by all the entities. Furthermore, the entire session needs to use the same shared secret, which undermines the security of the communication as well.

Among the top-down approach schemes, the only solution that supports incremental deployment is mbTLS [29]. If the server does not support mbTLS, the client first establishes a standard TLS session with the server. Then, the client sends the segment keys to each middlebox that does support mbTLS. To achieve individual secrecy, the client generates the different segment keys for all the segments and distributes keys to the corresponding middleboxes (two segment keys per one middlebox), which is inefficient.

Audit Mechanisms: We propose three audit mechanisms for the clients to audit middleboxes while performing an maTLS session: *Explicit Authentication*, *Security Parameter Verification*, and *Valid Modification Checks*. These mechanisms necessitate some data structures for middleboxes, such as signatures or message authentication codes (MACs), to demonstrate accountability for every message. We prefer to use MACs, as signatures require higher computation overhead on their generation. Thus, entities will use hash-based message authentication codes (HMACs) when signatures are not necessary. To this end, we introduce *accountability keys* that are to be used as HMAC keys. The accountability key is established between the endpoints and middleboxes; thus, each middlebox should establish one accountability key with each endpoint (two in total), while the client and the server each need one accountability key for each middlebox, and share one more key between them.

We overview the audit mechanisms in Table I, alongside their notation in Table II.

(1) *Explicit Authentication* guarantees authentication of the server as well as the middleboxes by validating received certificates. If there are any suspicious middleboxes, the maTLS session can be aborted. The server sends its certificate in the `ServerCertificate` message during the maTLS handshake. Whenever the middleboxes receive this message, each of them simply appends its certificate, so that the client can receive all the certificates up to the server. As the client receives all the certificates, she does not need to worry about the degradation of certificate-level due to forged certificates by middleboxes. Similarly, DANE or CT can also be supported with middleboxes.

When receiving a sequence of certificates, the client should validate all of the certificates as well as recording the order of the certificates, up to the server.

(2) *Security Parameter Verification* allows the client to audit the security association of each maTLS segment, and to confirm the accountability keys as well as their order. To this end, the middleboxes have to present the security parameters (of each segment), that is, the chosen TLS version, the negotiated ciphersuite, the hashed master secret (i.e., , $H(ms)$), and a (hashed) transcript of the TLS handshake (i.e., the `verify_data` in the `Finished` message). The selected TLS version and ciphersuite show the degree of confidentiality of the corresponding maTLS segment. The hashed master secret demonstrates the uniqueness of segment keys. The transcript, a digest of handshake messages in the maTLS segment, is used to prove that two entities involved in the segment performed the handshake without any modification by an attacker.

However, middleboxes could potentially give false information to the client. To avoid such misbehavior, we propose a *security parameter block* – an unforgeable cryptographic proof of security information for each segment. Each block contains the security parameters and their

Notation	Meaning
C	Client
S	Server
MB_i	ith Middlebox in the session ($1 \leq i \leq n-1$)
e_i	ith Entity in the session where ($e_0 = C, e_n = S$)
$segment_{i,j}$	The maTLS segment between e_i and e_j
m	Message
m_i	Message sent from e_i
$a\|\|b$	a concatenated with b
$PRF(a,b,c)$	Pseudorandom function in [9] to derive keys ($a:\ secret, b:\ label, c:\ seed$)
$Sign(k,m)$	Signature function on m with a key k
$H(m)$	Hash function on m
$Hmac(k,m)$	Keyed hash-based MAC function with a key k on m
$Ae(k,m)$	Authenticated encryption on m with a key k
(sk_e, pk_e)	Entity e's (secret key, public key) pair
$Cert_e$	Entity e's certificate
ID_{e_i}	Identity of e_i. $ID_{e_i} = H(pk_{e_i})$
g	Generator of a DH group
(a, g^a)	Ephemeral DH key pair
$p_{i,j}$	Security parameters that includes the negotiated version, the negotiated ciphersuite, the hashed master secret, and the transcript between e_i and e_j
$ak_{e,f}$	Accountability key of e established with f (We simply write ak_e when f is fixed in the context)
$HMAC_e$	The result of $Hmac(k,m)$ by e
ML_e	Modification log generated by e

TABLE II: Notation used in this paper

HMAC value. The two entities of a maTLS segment, say $segment_{i,i+1}$, present the security parameters of the segment, respectively for cross-verification.

All the entities except the client in the maTLS session generate the security parameter block. The basic structure of the block is in the form of:

$$ID_{e_i}||p_{i,i+1}||Sign(sk_{e_i}, Hmac(ak_{e_i}, p_{i-1,i}||p_{i,i+1}))$$

One entity e_i first generates an HMAC over the security parameters in its two segments, namely $segment_{i-1,i}$ and $segment_{i,i+1}$, and signs on the resultant HMAC. Then, e_i prepends its identifier and the security parameters of the segment in the direction of the server with the signature. When the block is generated, e_i forwards it toward the client.

For a server ($S = e_n$) that is only involved in one segment, i.e., $segment_{n-1,n}$, the server sends $ID_{e_n}||Sign(sk_{e_n}, Hmac(ak_{e_n}, p_{n-1,n}))$ in which the term corresponding to $p_{i,i+1}$ in the above expression is removed.

When the client receives a series of security parameter blocks, it can confirm all security parameters negotiated between each entity by verifying the signature of signed HMACs. Verification fails could be due to modified security parameters, missing or incorrect order of the middleboxes; thus the client must abort the negotiation process. Once the client can successfully verify all the security parameters, accountability keys, the order of the middleboxes in the maTLS session, it can further decide whether to accept the session based on its policy. For example, the client might abort the connection if any of the segments is established with a weak algorithm such as an RC4 [36].

(3) *Valid Modification Check* allows a client to audit which entity has modified the message. When an entity forwards a message to the next entity it also generates a cryptographic proof, called a *modification log* (ML). Basically, it is to compare the incoming and outgoing message from the entity by attaching (1) a HMAC generated from both received and sending message using its accountability keys (ak_e), (2) a digest of the received message ($H(m_{i+1})$), and its identifier (ID_{e_i}). Assuming that the message is coming from the server (e_n) to the client (e_0), we can define the ML generated from the e_i, which is denoted as ML_{e_i}:

$$ID_{e_i}||H(m_{i+1})||Hmac(ak_{e_i}, H(m_i)||H(m_{i+1}))||ML_{e_{i+1}}$$

Here, we can apply some optimization techniques to reduce the size of the MLs in specific scenarios. First, the server does not need to append the $H(m_n)$ in the ML; the first middlebox to the server (e_{n-1}) can simply calculate the $H(m_n)$. Also, the server does not have a prior message, thus the ML_{e_n} can be defined as $ID_{e_n}||Hmac(ak_{e_n}, H(m_n))$. Second, when an entity (e_i) does not modify any message (i.e., read-only middlebox), we can further reduce the size of the ML_{e_i} by (1) simply generating a $HMAC_i$ from the previous $HMAC_{i+1}$ and (2) omitting its digest ($H(m_i)$) and even its ID (ID_{e_i}). Thus, if the client detects a missing ID while parsing the series of MLs, it can assume that the message has not been modified among the middleboxes with the missing IDs. For example, if an entity (e_i) receives a message that has never been modified, the ML that the entity received will be

$$ID_{e_n}||Hmac(ak_{e_{i+1}}, Hmac(ak_{e_{i+2}}, \cdots, Hmac(ak_{e_n}, H(m_n))),$$

Once e_i modifies the message, however, the ML produced from e_i will be

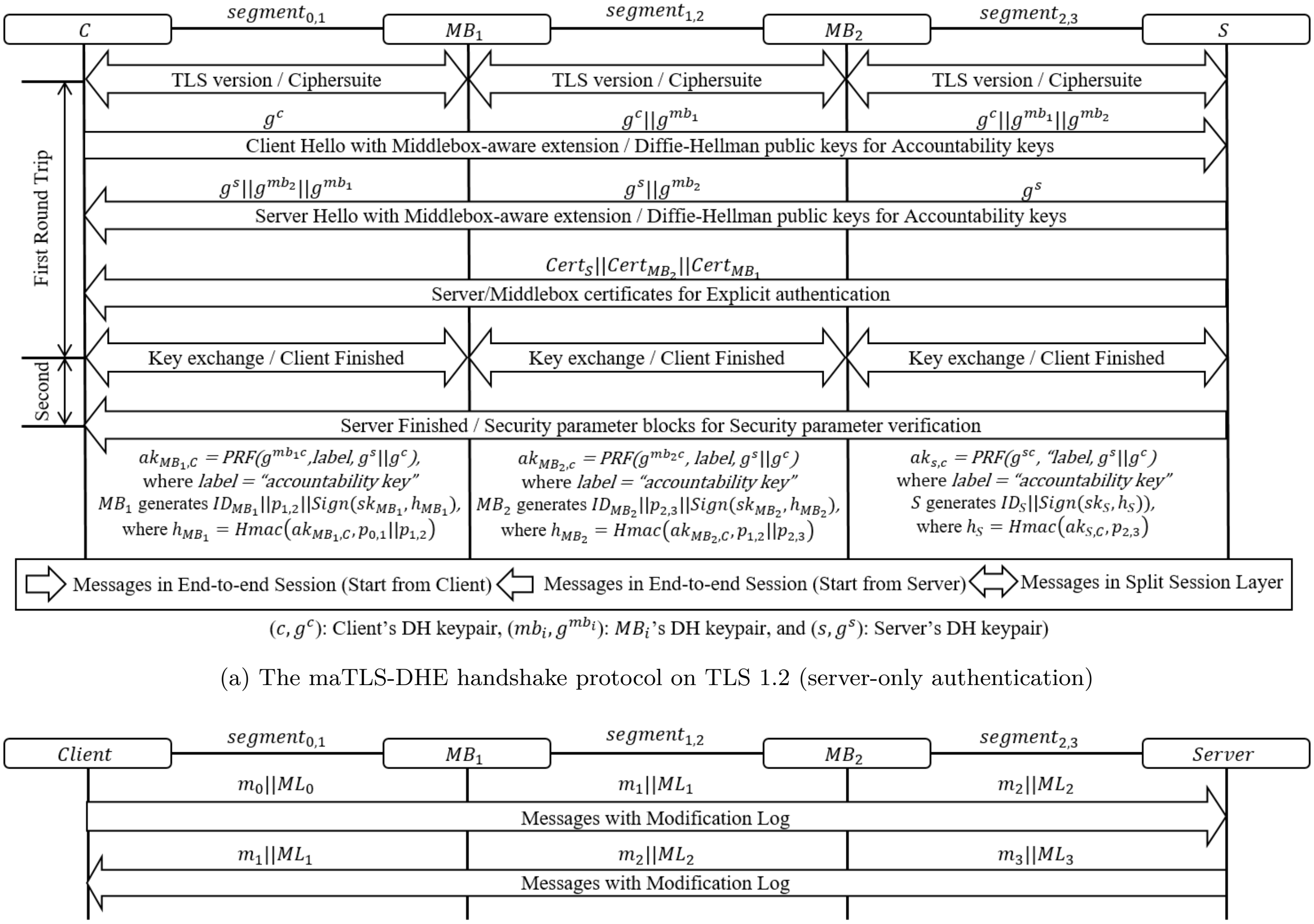
$$ID_{e_i}||H(m_i)||Hmac(ak_{e_i}, H(m_i)||H(m_{i+1}))||ID_{e_n}||HMAC_{e_{i+1}}$$

which implies that the message between the middlebox e_{i+1} and e_n has never been modified.

Once the receiver (i.e., the client in this example) obtains the series of MLs, it can extract the digests of all the modified messages, track the identifiers of the middleboxes that performed the write operation, and finally verify each ML using its HMAC.

C. maTLS Handshake Protocol

A client performs a maTLS handshake to negotiate accountability keys, to authenticate the server and middleboxes, and to perform security parameter verification. The maTLS handshake protocol, which extends TLS 1.2, is shown in Figure 3a. In the first round-trip, the client expresses its preference to perform the maTLS protocol by adding the *Middlebox_Aware* extension to the `ClientHello` message. The client generates its DH key pair (say, (a, g^a)) and inserts the DH public key (g^a) into the extension. Then, the client sends the `ClientHello` message with the highest possible TLS version and a set of supporting ciphersuites. On receiving the `ClientHello`, each middlebox finds the client's maTLS extension, generates its own DH key pair, and extracts the list of DH public keys from the maTLS extension. After that, it appends its own DH public key, and sends the new `ClientHello` with the DH public keys toward the client's intended server.


$segment_{0,1}$
$segment_{1,2}$
$segment_{2,3}$
C
MB_1
MB_2
S
First Round Trip
Second
TLS version / Ciphersuite
TLS version / Ciphersuite
TLS version / Ciphersuite
g^c
$g^c||g^{mb_1}$
$g^c||g^{mb_1}||g^{mb_2}$
Client Hello with Middlebox-aware extension / Diffie-Hellman public keys for Accountability keys
$g^s||g^{mb_2}||g^{mb_1}$
$g^s||g^{mb_2}$
g^s
Server Hello with Middlebox-aware extension / Diffie-Hellman public keys for Accountability keys
$Cert_S||Cert_{MB_2}||Cert_{MB_1}$
Server/Middlebox certificates for Explicit authentication
Key exchange / Client Finished
Key exchange / Client Finished
Key exchange / Client Finished
Server Finished / Security parameter blocks for Security parameter verification
$ak_{MB_1,C} = PRF(g^{mb_1c}, label, g^s||g^c)$,
where $label$ = "accountability key"
MB_1 generates $ID_{MB_1}||p_{1,2}||Sign(sk_{MB_1}, h_{MB_1})$,
where $h_{MB_1} = Hmac(ak_{MB_1,C}, p_{0,1}||p_{1,2})$
$ak_{MB_2,c} = PRF(g^{mb_2c}, label, g^s||g^c)$
where $label$ = "accountability key"
MB_2 generates $ID_{MB_2}||p_{2,3}||Sign(sk_{MB_2}, h_{MB_2})$,
where $h_{MB_2} = Hmac(ak_{MB_2,C}, p_{1,2}||p_{2,3})$
$ak_{s,c} = PRF(g^{sc}$, "label, $g^s||g^c)$
where $label$ = "accountability key"
S generates $ID_S||Sign(sk_S, h_S))$,
where $h_S = Hmac(ak_{S,C}, p_{2,3})$
Messages in End-to-end Session (Start from Client)
Messages in End-to-end Session (Start from Server)
Messages in Split Session Layer
(c, g^c): Client's DH keypair, (mb_i, g^{mb_i}): MB_i's DH keypair, and (s, g^s): Server's DH keypair)
$Client$
$Server$
$m_0||ML_0$
$m_1||ML_1$
$m_2||ML_2$
Messages with Modification Log
$m_1||ML_1$
$m_2||ML_2$
$m_3||ML_3$
Messages with Modification Log

(a) The maTLS-DHE handshake protocol on TLS 1.2 (server-only authentication)

(b) The maTLS record protocol with a modification log.

Fig. 3: **The maTLS protocol.** The maTLS handshake protocol is responsible for explicit authentication and security parameter verification, while the maTLS record protocol executes valid modification checks.

This process is repeated at every middlebox on the way to the server.

The server generates its own DH key pair (say, (b, g^b)) and sends the `ServerHello` message with the DH public key (g^b) and the selected TLS version and ciphersuite for the maTLS segment. On receiving `ServerHello`, each middlebox processes the message as the middlebox do on `ClientHello` and determines the TLS version and the ciphersuite to be used in the maTLS segment.

Then, each entity negotiates the TLS version and the ciphersuite with its neighbor entity for each maTLS segment. Furthermore, both endpoints receive the DH public keys from all entities and each middlebox has two DH public keys (i.e. the client's and the server's). With their own DH private keys, all entities generate the accountability keys by using the `PRF` function defined in [9] with the server's DH public key and the client's DH public key as seeds. For a `label`, one of the input parameters of the `PRF` function, we use the string, "accountability key."

The `ServerCertificate` message is sent after the `Hello` messages. The server sends its own certificate and each middlebox appends its middlebox certificate. The client performs explicit authentication in order to accept the server and the middleboxes. Then, the client maps each accountability key to the corresponding identity, where an identity is a digest of an entity's public key. Although the server does not receive the certificates, the server can identify the client from the accountability key.

After receiving the certificates, each maTLS segment exchanges key materials via the `ServerKeyExchange` and `ClientKeyExchange` messages. Using the key material, all entities generate shared secrets of the segment.

Finally, `Finished` messages are exchanged to verify the handshake between two peers in each segment, followed by a newly defined `ExtendedFinished` message that includes security parameter blocks from the server to the client. The client performs security parameter verification and confirms the proofs of private key possession by verifying the signatures by processing the `ExtendedFinished` message.

D. maTLS Record Protocol

The maTLS record protocol provides data source authentication, modification accountability, and path integrity during data exchange. The maTLS record protocol is illustrated in Figure 3b. For each message, the record protocol generates the data source, initializes an ML, and inserts its source MAC. On receiving the message and its ML, each middlebox processes the ML as mentioned earlier. A read-only middlebox extracts the final HMAC from the ML, performs the HMAC operation over the previous HMAC to put its fingerprint, and updates the MAC. A writer middlebox appends the modification MAC to the ML.

Upon receipt of the message, the destination performs valid modification checks by validating the ML, aborting the connection if there has been an invalid modification by middleboxes. The destination also verifies the source of the incoming message; for example, a server can abort the connection if the HTTP request message (over maTLS) did not originate from the client. Furthermore, since all the middleboxes in the session leave their own MACs in the ML whenever the data is passed the middleboxes, the endpoints can confirm whether the order of the middleboxes is preserved by verifying the MACs with the accountability keys in sequence.

VI. Security Verification

We analyzed the security goals of the maTLS protocol using Tamarin [24], an automated verification tool. Tamarin is built upon a multiset rewriting model, which supports the unbounded analysis of security protocols based on a robust equational theory. Tamarin is capable of accurately modeling Diffie-Hellman style key exchange, and is built upon the Dolev-Yao adversary.

The Tamarin execution model observes the development of a series of *states*, each of which is a multiset of *facts*. Each fact represents a detail about the current execution: for example, the $Out(msg)$ fact indicates that the message msg has been sent out to the communication network, while the fact $Ltk(A, k)$ might represent that the agent A has a long-term encryption key k. Facts are added and removed from the state through a series of user-defined *rules*, each of which is denoted by a triple `l` → `[ a ]` → `r`. Here, `l`, `a`, and `r` are collections of facts — for the rule to execute, the facts `l` are removed from the state and replaced by the facts `r`. The facts `a` form a *trace*: an indelible history of event markers that describe the progression of the protocol's execution.

Security goals, named lemmas, are expressed as first-order logic formulae describing requirements on the existence and ordering of certain events, usually quantified over all possible executions. If a formula is violated (generally indicating that a goal has not been met), Tamarin generates a graph showing a trace that leads to the contradicting state.

A. Protocol Rules

The protocol rules for maTLS can be divided broadly into three categories. The first handles the *setup* rules of the protocol. These represent events such as the registration of server or middlebox certificates. Second, a set of *corruption* rules describe the main ways in which an agent may violate their specification — for example, giving their long-term private key to the adversary. Finally, the *protocol* rules describe the actual actions of the participants. The protocol rules are again divided into two parts, namely *Handshake* rules and *Communication* rules, to capture the maTLS handshake protocol and the maTLS record protocol, respectively.

B. Security Claims

With the protocol rules, we modeled the core security goals of maTLS. We formally describe our security goals in the form of the first order logic formulae, examples of which are shown in Table III. Note that the goals shown in the table are slight simplifications of those in the full analysis (for example, they must be taken modulo corruption).

The results of the analysis show that the maTLS protocol satisfies the core security goals.[6]

VII. Evaluation

A. Experiment Settings

To demonstrate the feasibility of the maTLS protocol, we implemented it using the OpenSSL library. Our testbed consists of a client (C), a client-side middlebox (MB_C), a server-side middlebox (MB_S), and a server (S)[7]. The server-side machine and server are equipped with an Intel Xeon CPU E5-2676 at 2.40GHz with 1GB memory. We used an Intel Xeon CPU E3-2650 at 2.30GHz and 64GB memory for the client-side middlebox, and an Intel i5-6600 CPU at 3.30GHz and 16GB PC for the client.

During our experiments, the client and the client-side middlebox were located on a campus network. We ran tests with the server (and the server-side middlebox) located at three different locations: in the same country (intra-country testbed), in different countries but the same region (intra-region testbed), and in different continents (inter-region testbed). The round-trip times between two entities in each scenario are shown in Table IV.

After establishing an maTLS session, the client requests an HTML page of 200 bytes with an HTTP GET message, terminating the connection after completing the download of the HTTP response. Each plotted value is the average of 100 measurements. We compare the performance overhead of maTLS with those of SplitTLS and mcTLS [30], the latter of which is the original protocol of TLMSP.

We used an ECDH key exchange algorithm over the secp256r1 elliptic curve for the accountability keys, the SHA256 function for the hash algorithm, and a SHA256-based ECDSA for the signature algorithm.

[6] The full Tamarin implementation can be found at our public repository at https://github.com/middlebox-aware-tls/matls-tamarin.

[7] The source code of the library as well as the test applications are available at https://github.com/middlebox-aware-tls/matls-implementation

Security Goal	Code Snippet	Description

Server Authentication	`All C S nonces #tc.`
	`  C_HandshakeComplete(C, S, nonces)@tc`
	`==>`
	`Ex #ts.`
	`  S_HandshakeComplete(C, S, nonces)@ts &`
	`  (#ts < #tc)`
		When a client believes she has finished a maTLS handshake, the corresponding server also believes he has established a session with the client, sharing the same accountability key data

Middlebox Authentication	`All C MB last next nonces #tc.`
	`  C_MB_HandshakeComplete(C, MB, last, next, nonces)@tc`
	`==>`
	`Ex #tmb.`
	`  MB_C_HandshakeComplete(C, MB, last, next, nonces)@tmb &`
	`  (#tmb < #tc)`
		When the client confirms a middlebox as part of the handshake, the client shares accountability key data with them

Segment Secrecy	`All C M S nonces params #tc #tcomplete.`
	`  C_ParameterVerification(C, M, nonces, params)@tc &`
	`  C_HandshakeComplete(C, S, nonces)@tcomplete`
	`==>`
	`Ex #tmb.`
	`  MB_SecurityParameters(C, M, nonces, params)@tmb &`
	`  (#tmb < #tc)`
		When the maTLS session is established, a client correctly verifies the security parameters used in each segment

Individual Secrecy	`All C S nonces #tc #tcomplete.`
	`  C_HandshakeComplete(C, S, nonces)@tcomplete`
	`==> (`
	`  All a1 a2 b1 b2 keyA keyB #tmb1 #tmb2.`
	`    SegmentKeyMade(a1, a2, nonces, keyA)@tmb1 &`
	`    SegmentKeyMade(b1, b2, nonces, keyB)@tmb2`
	`  ==> (`
	`    not (keyA = keyB) |`
	`    (a1 = b1 & a2 = b2)`
	`  ))`
		At the end of a maTLS handshake, each segment has established distinct TLS keys

Data Authentication	`All C S nonces req resp #trecv.`
	`  C_BelievesSentFromServer(C, S, nonces, req, resp)@trecv`
	`==>`
	`Ex #tresp.`
	`  ServerSent(C, S, nonces, req, resp)@tresp`
		When a client receives a message during the maTLS record phase, the hash value from the server is a faithful digest of the original message

Modification Accountability	`All C S nonces req #trecv.`
	`  ReceiveResponse(C, S, req, nonces)@trecv`
	`==> (`
	`  All before after M #tc.`
	`    ModificationChecks(C, M, req, nonces, before, after)@tc &`
	`    ( #tc < #trecv ) | (#tc = #tcrecv)`
	`  ==> (`
	`  Ex #tmb.`
	`    MB_Modification(C, M, req, nonces, before, after)@tmb  &`
	`    #tmb < #tc`
	`    ))`
		When an endpoint receives a message during the maTLS record phase, the agent believes that a middlebox has changed the message if and only if that middlebox did make a change

Path Integrity	`All a1 a2 a3 nonces #ta #tb.`
	`  PathOrderingEstablished(nonces, a1, a2)@ta &`
	`  PathOrderingEstablished(nonces, a2, a3)@tb`
	`==> (`
	`  All id #tf. ForwardAction(nonces, id, a2, a3)@tf`
	`    ==> (`
	`      Ex #tp. ForwardAction(nonces, id, a1, a2)@tp &`
	`      #tp < #tf`
	`  ))`
		The client knows the order of the intermediate middleboxes in an maTLS session. Messages will always travel in this order.

TABLE III: **Security Lemmas.** Tamarin representations of the core security goals of the maTLS handshake and record phase protocols. For the full specifications, see our git repository.

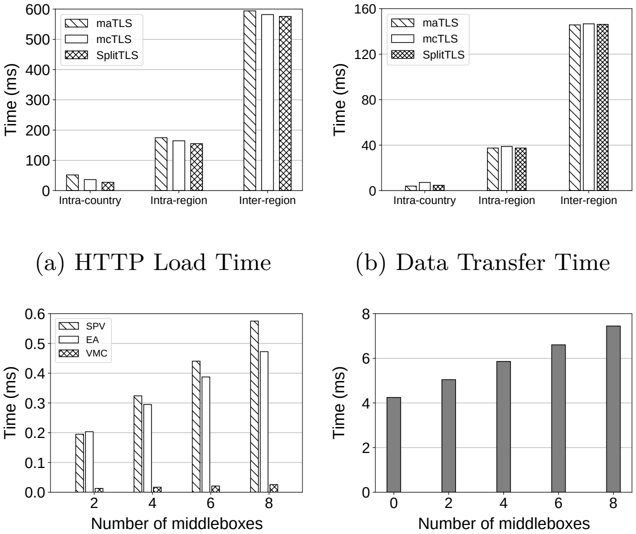
Testbed	C-MB_C	MB_C-MB_S	$MB_S - S$
Intra-country	0.610ms	4.308ms	0.617ms
Intra-region	0.610ms	34.864ms	0.524ms
Inter-region	0.610ms	141.467ms	0.569ms

TABLE IV: **Networking Settings.** The round-trip times between two points in each scenario are shown, where C and MB_C are in the same campus, and MB_S and S are in the same data center.

B. HTTPS Page Load Time

We first evaluate the time elapsed to fetch a 200 byte file from the server in the maTLS protocol, which is compared with the SplitTLS and mcTLS protocols. Figure 4a summarizes the time taken from starting a TCP handshake to finishing the download of the last byte of the content. We observe that the maTLS protocol introduces a slight delay (15.58ms – 24.40ms) compared to mcTLS and SplitTLS in the general case.

We believe this is mainly due to the message order dependency in maTLS. Unlike SplitTLS, where each TLS segment is established *completely independently*, the maTLS segments are established piecewise *sequentially* as some signaling messages (e.g., `ClientHello, ServerHello, ServerCertificate`) must be exchanged between the client and the server through the middleboxes in sequence. Thus, in maTLS, each middlebox needs to wait until these messages arrive while performing the handshake.



(a) HTTP Load Time

(b) Data Transfer Time

(c) Integrity Verification Time

(d) CPU Processing Time

Fig. 4: Numerical results reveal that maTLS incurs slightly more delay, ranging from 15.58ms to 24.40ms against mcTLS and SplitTLS, mainly due to the signature verification and key generation needed in the maTLS handshake. (EA: Explicit Authentication, SPV: Security Parameter Verification, VMC: Valid Modification Checks)

To quantify the overhead that the maTLS record protocol requires, Figure 4b shows the data transfer time, which starts at the client sending an HTTP GET (a single packet) and ends at the client receiving an HTTP RESPONSE (a single packet). Interestingly, we notice that the overhead of the maTLS record protocol is marginal. For example, in the intra-region testbed scenario, the data transfer time is 37.42ms, 37.41ms, and 38.82ms in maTLS, SplitTLS, and mcTLS respectively.

From Figures 4a and 4b, we conclude that the maTLS overhead is mainly due to the setup of a maTLS session, which implies that once the session is established, maTLS provides similar performance to the others while *preserving all security merits that we have discussed.*

C. Scalability of Three Audit Mechanisms

Next, we evaluate the scalability of the maTLS audit mechanisms: Explicit Authentication (EA), Security Parameter Verification (SPV), and Valid Modification Checks (VMC). Note that the number of required HMAC operations increases in proportion to the number of the middleboxes. Thus we now wish to check the scalability of the HMAC operations in maTLS for its feasibility. To this end, we increase the number of middleboxes in the same data center to quantify the computational overhead due to the audit mechanisms by measuring the validation time for each arriving packet (Figure 4c).

We observe that the overhead of the three audit mechanisms is almost negligible. For example, it takes 0.195ms to verify security parameter blocks, 0.203ms to validate certificates, and 0.013ms to check the modification record for two middleboxes. Also, we observe that the overhead increases *linearly* with the number of middleboxes; for each incoming packet, only an extra 0.045ms and 0.063ms overhead is required for the explicit authentication checks and security parameter verification, respectively. It is worth noting that the delay of explicit authentication is mainly due to certificate validation, which accounts for around 95% of the delay. Likewise, signature verification accounts for more than 91% of the delay of the security parameter verification. The overhead for valid modification checks is marginal as it uses HMAC operations to verify the ML, which turns out to be only 0.026ms, even with 8 middleboxes. We believe that the auditing mechanisms of maTLS can achieve their goals without incurring a substantial delay.

D. CPU Processing Time

Next, we evaluate the CPU processing time for a maTLS handshake as the number of middleboxes increases. We place all the middleboxes and the endpoints in the same data center to minimize the impact of networking delay. As shown in Figure 4d, the CPU processing time for the maTLS handshake also linearly increases by on average 0.398ms for each middlebox. This increment is mainly due to the multiplication operations required to add an ECDH shared secret, and generating accountability keys using a `PRF`, which account for 0.367ms (92.2% of the increment) and 0.016ms (4.0% of the increment), respectively.

VIII. Discussions

A. Incremental Deployment

The maTLS protocol can be executed even if not all the entities support it. In other words, a session can have both maTLS segments and TLS segments at the same time. For example, when a client and two middleboxes support maTLS and the server does not, maTLS segments can be set up between the client and the two middleboxes. In this case, the middlebox farthest from the client in the maTLS segments establishes a standard TLS segment with the server. Following the maTLS protocol, all the middleboxes in the maTLS segments send their own certificate to the client. Therefore, the client will receive a bundle of middlebox certificates, but not the certificate including the server's name. This will cause the client to issue a warning message.

To resolve the problem, we require that the farthest middlebox in the maTLS segments should send not only its middlebox certificate but also the received certificate from the standard TLS segment. This allows the client to receive the server's certificate and thus validate it. Unfortunately, this requires that the client must trust that the middlebox sent the certificate that it received, and correctly validated the server certificate in the standard TLS handshake. However, the client can still authenticate the participating middleboxes and verify their security parameters, which is not be supported by the current practice.

B. Abbreviated Handshake

maTLS supports abbreviated handshakes using session IDs/tickets in TLS 1.2, or pre-shared keys in TLS 1.3, which need not extend the handshake. A client can resume a maTLS session using the abbreviated handshake protocol. The middlebox (closest to the server) can resume its maTLS segment with the server, as it knows the session ID, pre-shared key, or session ticket. The middlebox, however, does not have the accountability key shared between the client and the server; thus, the server is able to detect incorrect session resumptions by verifying the modification log if an adversary attempts to impersonate the middlebox.

C. Mutual Authentication

Like the standard TLS protocol, maTLS also supports mutual authentication by sending a `CertificateRequest` message to the client during the TLS handshake. In this case, the client also sends her certificate upon receipt of the `CertificateRequest` message from the server. The middleboxes can simply append their certificates to her certificates while being forwarded to the server so that both the client and the server authenticate each other's certificates. After that, the client and the server each send a `ExtendedFinished` message to verify the possession of their private keys.

D. TLS 1.3 Compatibility

TLS 1.3 [12] has been recently approved and is expected to be widely deployed. The maTLS protocol can support TLS 1.3 by adding a `ExtendedFinished` message after a server's `Finished` message in the server-only authentication mode. The only difference is that TLS 1.2 requires two round-trips for session establishment, while TLS 1.3 only requires one and a half round trips. Unfortunately, this means that individual segments running TLS 1.2 will negate some of the speed-up benefits from TLS 1.3.

IX. Related Work

A. Discussion on Middleboxes

Studies on the SplitTLS practice: Frack *et al.* [4] showed that content providers sharing a private key with a hosting provider (such as CDNs) may significantly affect the security of the HTTPS ecosystem; an attacker who compromises ten hosting providers is estimated to obtain the control of 45% of all content providers. Lin-Shung *et al.* [17] demonstrated that there were a large number of forged certificates in the wild, most of which were generated by client-side middleboxes. They also showed that these certificates can be used to trick victims, who had installed the root certificates of the forged certificates.

Debates on Explicit middleboxes: There have been two IETF drafts that highlight the problems with HTTPS middleboxes and propose new design principles. Both Nottingham [32] and Narayanan [27] emphasize that endpoints should be aware of middleboxes, and that their modifications on the messages should be detectable.

B. Proposals

TLS extensions: Several proposals have been made to extend the TLS protocol to support middleboxes.

(1) `Explicit Trusted Proxy [22]:` This work proposes that middleboxes should have their own certificates for authentication. Each middlebox certificate should be an EV certificate with proxyAuthentication value in the ExtendedKeyUsage field. This makes middleboxes visible with their certificates; however, endpoints can only authenticate the immediately adjacent middleboxes, and cannot get any information about the other middleboxes.

(2) `TLS Keyshare extension [31]:` In this protocol, the client initiates a TLS handshake by sending information about authorized middleboxes to the server. During the handshake, the middleboxes inspect the TLS handshake message and notify the endpoints of any unsupported ciphersuites. After the session is established by the endpoints, the authorized middleboxes receive the session key from the endpoints, allowing them to perform their functionality. Since the same key is shared across all the segments, the keystream is reused, which weakens overall security. Furthermore, this work does not consider modification-related properties.

(3) `TLS ProxyInfo extension [45]:` Each split segment is separately established, as in the maTLS protocol. All the middleboxes pass their certificates and negotiated security parameters with their signatures to the endpoints, who can authenticate all the middleboxes and confirm security parameters. However, in this protocol, the endpoints must blindly trust the information about each segment from each middlebox. Furthermore, data source authentication, modification accountability, and path integrity are not considered.

(4) `Multi-context TLS (mcTLS) [30]:` mcTLS aims to restrict the behavior of middleboxes by applying the least privilege principle. Endpoints generate two MAC keys for middleboxes: read and write. If a middlebox is authorized to read and write, it obtains both MAC keys. If it can only read the TLS traffic, it gets only the read MAC key. All the middleboxes are authenticated from their certificates. However, as mcTLS uses one session key, it undermines the security of the session if any of middleboxes involved is a writer. Furthermore, after modification by a writer middlebox, the receiver cannot know who has sent the data.

(5) `Transport Layer Middlebox Security Protocol (TLMSP):` TLMSP is an improved version of mcTLS, which is being standardized in ETSI. Based on mcTLS, it optionally introduces an audit trail that records each middlebox's inbound HMAC and outbound HMAC to check the modification by the middlebox and the order of the middleboxes in the chain. However, TLMSP uses a top-down approach, which is not suitable for incremental deployment due to the reasons described in Section V-B.

(6) `Middlebox TLS (mbTLS) [29]:` mbTLS allows outsourced middleboxes to participate in a TLS session. mbTLS extends TLS for middleboxes running on Intel

SGX technology. When endpoints perform a TLS handshake with each other, each endpoint opens a secondary TLS session with all the middleboxes that it leverages for remote attestation. After the primary TLS handshake, the endpoints send the session key to each of the middleboxes. However, mbTLS offers no information about each segment's secrecy or changes to TLS traffic. Instead, the endpoints rely on the TEE technology to assure middlebox integrity.

Cryptographic approaches: `BlindBox` [41] and `Embark` [20] allow a monitoring gateway (in the client's network) to read TLS traffic without revealing its content to middleboxes on a third party cloud. To this end, they introduce a secondary channel using a special encryption technique (such as searchable encryption or order-preserving encryption). The client communicates with the server over a TLS session, and delivers the packets to the middleboxes via the secondary channel before the client sends packets to the server. Private data are not leaked to the middleboxes in these proposals, but they have two main drawbacks. First, the possible functionality of middleboxes is limited by the encryption techniques. Second, they require another round trip to middleboxes over the secondary channel before sending the data to the other endpoint.

TEE approaches: `SafeBricks` [35], `ShieldBox` [43], and `SGX-Box` [16] focus on guaranteeing security and protecting privacy (from middleboxes) by building middleboxes over TEE technology. The three schemes have different properties. For example, `SafeBricks` aims to apply the least privilege principle to middleboxes by using a type-safe language. `ShieldBox` seeks to supports syscalls in an enclave, and `SGX-Box` offers programmability to middlebox developers for easy deployment.

X. Conclusion

In this paper, we propose middlebox-aware TLS, dubbed maTLS, that allows middleboxes to participate in TLS networking in a visible and accountable fashion. The maTLS protocol seeks to achieve the following security goals: server authentication, middlebox authentication, path secrecy, data source authentication, and modification accountability, which are not comprehensively solved by the related work. To this end, maTLS relies on multiple mechanisms such as middlebox certificates, middlebox Transparency, security parameter blocks, and modification records to make middleboxs visible and auditable. We also analyze the security properties of the maTLS protocol using Tamarin, which formally proves that maTLS satisfies those goals. Furthermore, testbed-based experiments show that maTLS accomplishes those goals with mostly marginal performance overhead. For instance, the additional delays against the SplitTLS and the mcTLS protocols are less than 25 ms, which incurs mainly due to the signaling overhead in a handshake. Numerical results also show that the maTLS protocol is scalable in terms of number of middleboxs.

Acknowledgement

The authors would like to appreciate the anonymous reviewers for their comments to improve our paper. Also, many thanks to David T. Naylor for his comments and advice. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2016-0-00160, Versatile Network System Architecture for Multi-dimensional Diversity)

References

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy: How diffie-hellman fails in practice," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2015.

[2] W. Ashford, "Privdog ssl compromise potentially worse than superfish," February 2015. [Online]. Available: http://www.computerweekly.com/news/2240241126/PrivDog-SSL-compromise-potentially-worse-than-Superfish

[3] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," in *Security and Privacy (SP), 2015 IEEE Symposium on.* IEEE, 2015, pp. 535–552.

[4] F. Cangialosi, T. Chung, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson, "Measurement and analysis of private key sharing in the https ecosystem," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2016, pp. 628–640.

[5] T. Chung, D. Choffnes, and A. Mislove, "Tunneling for transparency: A large-scale analysis of end-to-end violations in the internet," in *Internet Measurement Conference (IMC)*, 2016.

[6] Comodo, "Comodo report of incident - comodo detected and thwarted an intrusion on 26-mar-2011," 2011. [Online]. Available: https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html

[7] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile," RFC 5280, Internet Engineering Task Force, May 2008, http://www.ietf.org/rfc/rfc5280.txt.

[8] X. de Carné de Carnavalet and M. Mannan, "Killed by proxy: Analyzing client-end tls interception software," in *Network and Distributed System Security Symposium*, 2016.

[9] T. Dierks, "The transport layer security (TLS) protocol version 1.2," 2008.

[10] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[11] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson, "The security impact of https interception," in *Network and Distributed Systems Symposium*, 2017.

[12] E. Rescorla, "The transport layer security (TLS) protocol version 1.3 (draft 28)," 2018. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-tls13-28

[13] Facebook, "Introducing our certificate transparency monitoring tool." [Online]. Available: https://www.facebook.com/notes/protect-the-graph/introducing-our-certificate-transparency-monitoring-tool/1811919779048165/

[14] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, “Measuring HTTPS adoption on the web,” in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017.

[15] Gaurang, “Nokia’s mitm on https traffic from their phone,” 2013. [Online]. Available: https://gaurangkp.wordpress.com/2013/01/09/nokia-https-mitm/

[16] J. Han, S. Kim, J. Ha, and D. Han, “Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module,” in *Proceedings of the First Asia-Pacific Workshop on Networking.* ACM, 2017, pp. 99–105.

[17] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson, “Analyzing forged ssl certificates in the wild,” in *Security and Privacy (SP), 2014 IEEE Symposium on.* IEEE, 2014, pp. 83–97.

[18] ITU-T RECOMMENDATION, “Information technology–open systems interconnection–the directory: Public-key and attribute certificate frameworks,” 2000.

[19] J. Jarmoc and D. Unit, “SSL/TLS interception proxies and transitive trust,” *Black Hat Europe*, 2012.

[20] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, “Embark: Securely outsourcing middleboxes to the cloud.” in *NSDI*, vol. 16, 2016, pp. 255–273.

[21] B. Laurie, A. Langley, and E. Kasper, “Certificate transparency,” Tech. Rep., 2013.

[22] S. Loreto, J. Mattsson, R. Skog, H. Spaak, G. Gus, and D. Druta, “Explicit trusted proxy in http/2.0,” 2012. [Online]. Available: https://tools.ietf.org/html/draft-loreto-httpbis-trusted-proxy20-01

[23] D. McGrew, D. Wing, Y. Nir, and P. Gladstone, “TLS proxy server extension,” 2012. [Online]. Available: https://tools.ietf.org/html/draft-mcgrew-tls-proxy-server-01

[24] S. Meier, B. Schmidt, C. Cremers, and D. Basin, “The tamarin prover for the symbolic analysis of security protocols,” in *International Conference on Computer Aided Verification.* Springer, 2013, pp. 696–701.

[25] D. Meyer, “Nokia: Yes, we decrypt your https data, but don’t worry about it,” 2013. [Online]. Available: http://gigaom.com/2013/01/10/nokia-yes-we-decryptyour-https-data-but-dont-worry-about-it/

[26] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, “X. 509 internet public key infrastructure online certificate status protocol-ocsp,” 1999.

[27] V. Narayanan, “Explicit proxying in http-problem statement and goals,” 2013. [Online]. Available: https://tools.ietf.org/pdf/draft-vidya-httpbis-explicit-proxy-ps-00.pdf

[28] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, “The cost of the s in https,” in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies.* ACM, 2014, pp. 133–140.

[29] D. Naylor, R. Li, C. Gkantsidis, T. Karagiannis, and P. Steenkiste, “And then there were more: Secure communication for more than two parties,” in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies.* ACM, 2017, pp. 88–100.

[30] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. López, K. Papagiannaki, P. R. Rodriguez, and P. Steenkiste, “Multi-context tls (mctls): Enabling secure in-network functionality in tls,” in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 199–212.

[31] Y. Nir, “A method for sharing record protocol keys with a middlebox in TLS,” 2012. [Online]. Available: https://tools.ietf.org/id/draft-nir-tls-keyshare-02.html

[32] M. Nottingham, “Problems with proxies in http,” 2014. [Online]. Available: https://tools.ietf.org/pdf/draft-nottingham-http-proxy-problem-01.pdf

[33] D. O’Brien, “Certificate transparency enforcement in google chrome,” 2018. [Online]. Available: https://groups.google.com/a/chromium.org/forum/#!msg/ct-policy/wHILiYf31DE/iMFmpMEkAQAJ

[34] M. O’Neill, S. Ruoti, K. Seamons, and D. Zappala, “TLS proxies: Friend or foe?” in *Proceedings of the 2016 Internet Measurement Conference.* ACM, 2016, pp. 551–557.

[35] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, “Safebricks: Shielding network functions in the cloud,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI’18), Renton, WA*, 2018.

[36] A. Popov, “Prohibiting rc4 cipher suites,” 2015. [Online]. Available: https://tools.ietf.org/html/rfc7465

[37] E. Rescorla, “HTTP over TLS,” 2000. [Online]. Available: https://tools.ietf.org/html/rfc2818

[38] J. Schlyter and P. Hoffman, “The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA,” 2012. [Online]. Available: https://tools.ietf.org/pdf/rfc6698.pdf

[39] T. J. Seppala, “New lenovo pcs shipped with factory-installed adware,” 2015. [Online]. Available: https://www.engadget.com/2015/02/19/lenovo-superfish-adware-preinstalled/

[40] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, “Making middleboxes someone else’s problem: network processing as a cloud service,” vol. 42, no. 4. ACM, 2012, pp. 13–24.

[41] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, “Blindbox: Deep packet inspection over encrypted traffic,” vol. 45, no. 4. ACM, 2015, pp. 213–226.

[42] SSLMate, “Cert spotter.” [Online]. Available: https://sslmate.com/certspotter/

[43] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, “Shieldbox: Secure middleboxes using shielded execution,” in *Proceedings of the Symposium on SDN Research.* ACM, 2018, p. 2.

[44] G. Tsirantonakis, P. Ilia, S. Ioannidis, E. Athanasopoulos, and M. Polychronakis, “A large-scale analysis of content modification by open http proxies,” in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[45] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson, “Internet x. 509 public key infrastructure (PKI) proxy certificate profile,” 2004.

[46] L. Waked, M. Mannan, and A. Youssef, “To intercept or not to intercept: Analyzing tls interception in network appliances,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security.* ACM, 2018, pp. 399–412.

[47] O. Williams, “Google dropping cnnic root ca after trust breach,” 2015, https://thenextweb.com/insider/2015/04/02/google-to-drop-chinas-cnnic-root-certificate-authority-after-trust-breach/.