공학박사 학위논문

Transport Layer Security Extensions for Middleboxes and Edge Computing

미들박스와 엣지컴퓨팅을 위한 전송 보안 계층 확장 연구

2020 년 8 월

서울대학교 대학원

컴퓨터공학부

이 현 우

Transport Layer Security Extensions for Middleboxes and Edge Computing

미들박스와 엣지컴퓨팅을 위한 전송 보안 계층 확장 연구

지도교수 권 태 경

이 논문을 공학박사 학위논문으로 제출함

2020 년 7 월

서울대학교 대학원

컴퓨터공학부

이 현 우

이현우의 공학박사 학위논문을 인준함

2020 년 6 월

위 원 장	최 양 희	(인)
부위원장	권 태 경	(인)
위 원	백 윤 흥	(인)
위 원	이 병 영	(인)
위 원	허 준 범	(인)

Abstract

Transport Layer Security Extensions for Middleboxes and Edge Computing

Hyunwoo Lee
Department of Computer Science and Engineering
The Graduate School
Seoul National University

Internet traffics are getting encrypted with HTTPS, which makes middleboxes blind. To introduce middleboxes in the encrypted session, the TLS interception schemes (a.k.a., SplitTLS) that abuse the public key infrastructure (PKI) are widely used in practice. Several papers, however, demonstrate that the SplitTLS practice is risky due to incorrect implementation or misconfiguration of middleboxes.

This dissertation aims to design secure and trustworthy methods to introduce middleboxes in the TLS session. To this end, we first classify middleboxes into two types called a middlebox-as-a-middlebox and a middlebox-as-an-middlebox. A middlebox-as-a-middlebox is an intermediary between a client and a server at communication time, while a middlebox-as-an-endpoint is an intermediary that takes on the role of a server during the session. An example of the former is an intrusion detection system and that or the latter is a web cache. Then we conduct literature reviews over 23 protocols (14 protocols for a middlebox-as-a-middlebox and 9 protocols for a middlebox-as-an-endpoint) that make middleboxes participate into TLS sessions.

From our reviews, we have learned the following lessons. For a protocol with a middlebox-as-a-middlebox, we should consider the least privilege of a middlebox to limit it not to perform

functionality with excessive permission in design. Also, since a server is involved into the session, we can use a server to help a client to understand a middlebox. For a protocol with a middlebox-as-an-endpoint, we should consider a method not to add further round-trips to a server. In addition, the number of secrets should be minimal and the overhead for the key management should not be placed on a server.

In this disseration, we propose two protocols called MATLS and TLS-SEED, based on our learnings.

The MATLS protocol is a protocol for a middlebox-as-a-middlebox. 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 these 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.

The TLS-SEED protocol is a protocol for a middlebox-as-an-endpoint, especially considering a scenario of edge computing. Edge computing is an emerging technology to bring computation and data storage closer to clients, to provide fast responses and to reduce the bandwidth usage in cloud servers. An edge computing platform is typically a third party to an application service provider and a client, both of which require high security assurance. Therefore, we propose TLS-SEED, a TLS extension that addresses risky private key sharing

and inefficient remote attestation on the third party, while preserving performance in edge computing. TLS-SEED allows an application service provider (i) to deploy its edge application without sharing its private keys, (ii) to authorize/deauthorize its edge application by performing remote attestation, while presenting sufficient information for a client to verify the edge application without relying on an attestation service. A central data structure of TLS-SEED is a CROSS CREDENTIAL (CC) that shows a client the trust relation between an application service provider and a trusted device. The CC also gives the client the ability to verify the integrity of the edge application. To formally analyze TLS-SEED, we introduce ACCE-SEED, a formal model for TLS-SEED, by extending the ACCE model for TLS, and show TLS-SEED is ACCE-SEED-secure. Furthermore, testbed-based experiments show that TLS-SEED can be substantiated with a negligible performance overhead.

Keywords: Transport Layer Security (TLS), Middleboxes, Edge Computing, Trusted Execution Environments (TEEs)

Student Number: 2015-21259

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Middleboxes are entities located in-between a server and a client to enhance security and performance in the session. For example, an intrusion detection system (IDS) monitors application data exchanged and blocks any malicious data. On the other hand, a web cache stores content on behalf of a server near a client and serves the content on request from a client.

Such middleboxes are widely deployed and are likely to be used more in the future. According to a survey from 57 enterprise networks [91], the number of middleboxes is similar to the number of L3 routers in those networks. Furthermore, enterprises that have large networks with more than 100k hosts spend one million dollars and above per year to deploy and to maintain middleboxes. Additionally, network appliance and security-as-a-service markets are getting bigger. For instance, a network appliance market evolves from 8.45 billion dollars (2018) to 13.97 billion dollars (2023) [65], while a security-as-a-service market is growing from 6.91 billion dollars (2018) to 16.92 billion dollars (2024) [37]. Also, a recent mobile technology, i.e., 5G or mobile edge computing, consider deploying edge

functions or virtualized network functions (VNFs) near base stations to perform services at user proximity [47].

Middleboxes, however, should address a challenge due to encryption as HTTPS (HTTP over TLS) [81] is widely adopted on the web[1]. Note that TLS makes middleboxes blind since it achieves end-to-end security; thus, middleboxes cannot read messages exchanged between a client and a server. Therefore, there have been much research that aims to address the problems.

Contribution Points. In this paper, we identify security properties for middleboxes and survey schemes used to enable middleboxes to function on encrypted messages. Specifically, we classify middleboxes into two categories, middlebox-as-a-middlebox (e.g., IDSes) and middlebox-as-an-endpoint (e.g., web caches), and conduct an extensive literature review on related protocols, based on the type of middleboxes respectively. From learnings that we get from our systemization, we propose MATLS and TLS-SEED, the former of which is relevant for a protocol with a middlebox-as-a-middlebox while the latter of which is proper to a middlbox-as-an-endpoint. Note that we conduct not only security analysis but also evaluation for two protocols.

Roadmap. The paper is organized as follows. We first present the background of middleboxes and introduce security issues in communicating with middleboxes (§1). We then review protocols that introduce middleboxes into encrypted sessions based on the type of middleboxes (§2). Next, we propose MATLS (§3) and TLS-SEED (§4). Finally, we present our concluding remarks (§5).

1.2 Background

In this paper, we define middleboxes as third-party entities that add functionalities to sessions to support a client or a server for the purpose of security and performance, while accessing to

[1] As time of writing, more than 72% of HTTP traffic is encrypted in Chrome from 10 countries, according to Google Transparency Report [32]

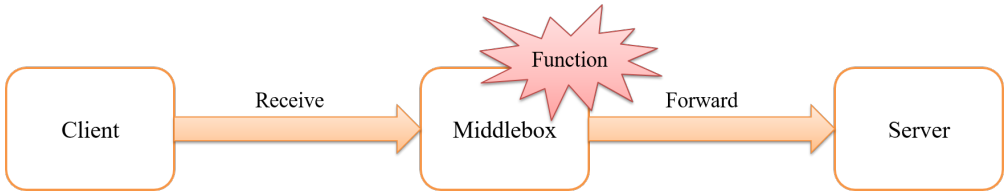


Figure 1.1: **Middlebox-as-a-Middlebox**. When a middlebox receives packets from one channel, a middlebox performs its functionality over packets and forwards them to the other channel. Examples are web application firewalls and anti-virus software. Note that a server is always involved in the session.

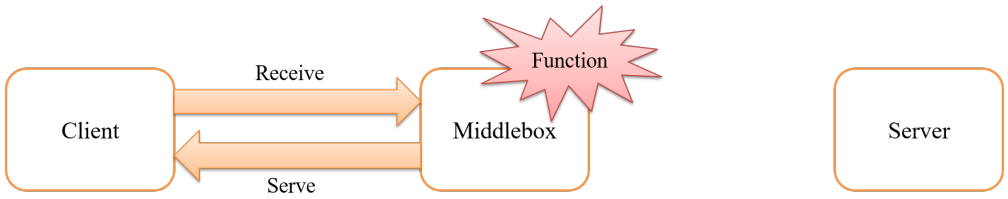


Figure 1.2: **Middlebox-as-an-Endpoint**. When a middlebox receives packets from one channel, a middlebox processes and replies to them. The other channel is used only in a limited case (e.g., pulling a content from a server). Examples are web caches or content delivery networks. Note that a server is not always involved in the session.

application data. Middleboxes can be classified into two categories that we call *middlebox-as-a-middlebox* (e.g., IDSes) and *middlebox-as-an-endpoint* (e.g., web caches) respectively. The current approach to make middleboxes participate in the encrypted session is called TLS interception (a.k.a., SplitTLS [45]) that is classified into *custom root certificate* and *private key sharing*. Note that much research [22, 25, 103, 73] have reported that it is vulnerable.

1.2.1 Types of Middleboxes

Middlebox-as-a-Middlebox. A *middlebox-as-a-middlebox* (as in Figure 1.1) is an intermediary between a client and a server at communication time. It receives packets from one channel, performs their functionality on packets, and forwards packets to the other channel. Examples of this type are intrusion detection systems, virus scanners, and application firewalls. Note that with a middlebox-as-a-middlebox, a server is always involved in the established session.

Middlebox-as-an-Endpoint. A *middlebox-as-an-endpoint* (as in Figure 1.2) is an intermediary that takes on the role of a server during the session. In other words, they serve requests as possible and thus a server is not always involved in the established session. A representative example of this type is an edge server in a content delivery network (CDN) that provides a client with content on behalf of a server.

Comparison. The remarkable difference between the two types is whether a server participates in the session. This is important since a server or a client can offer a cryptographic proof of a middlebox-as-a-middlebox to the other point at the communmication time, while it is impossible with a middlebox-as-an-endpoint.

1.2.2 Transport Layer Security

The TLS protocol [23, 82], 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 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.

1.2.3 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 [41] 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 [38] is used to allow alternative names of the certificate holder.

1.2.4 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 [19, 106].

To mitigate the risks from CA compromises, Google introduced the Certificate Transparency (CT) system [52], 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 [72]. 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 [96] and Facebook's CT Monitor [28] monitor each log server and alert a domain owner if a new certificate that binds to her domain name has been issued.

1.2.5 TLS Interception

Custom Root Certificate. The *custom root certificate* abuses centralized trust in public key infrastructure (PKI). A middlebox generates a private key and the corresponding certificate, which is installed into a trusted certificate store of a client. Whenever a client initiates a TLS handshake with an intended target name (say, `www.alice.com`), a middlebox generates a forged certificate signed with the custom root certificate and sends it to a client. Since a client trusts the custom root certificate, a middlebox succeeds to impersonate a target server and a client finally believes that she establishes a TLS session with a server (not a middlebox). This method is widely used in virus scanners [22] and network appliances [25, 103].

Private Key Sharing. The *private key sharing* requires a server to share his private key and certificate with a middlebox. A middlebox delivers a server's certificate when a client begins with a TLS handshake toward a server. Since the certificate sent by a middlebox is a genuine certificate of a server, a client cannot but believe that she is communicating with a server (not a middlebox) and finally establishes a TLS session with a middlebox. Content delivery networks (CDNs) [16] or cloud-based middlebox services such as security-as-a-service generally require a user to upload her private key and certificate.

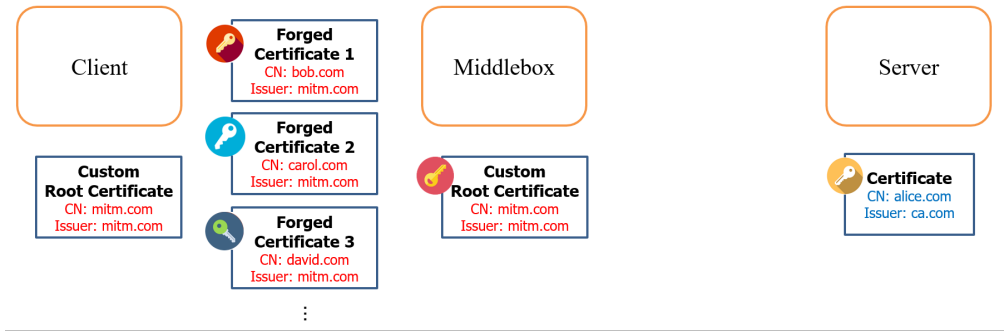


Figure 1.3: **Problems of Key Management in SplitTLS: Custom Root Certificate.** A private key that corresponds to a custom root certificate should be well protected. This is because if a private key that corresponds to a custom root certificate is leaked by an adversary, an adversary can impersonate arbitrary domains to a limited number of clients that trust the custom root certificate.

1.2.6 Problems of SplitTLS

We summarize problems of SplitTLS in terms of key management and communication below.

Key Management. Private keys in both custom root certificate and private key sharing should be securely managed. This is because if an adversary learns any of private keys, the situation would be catastrophic.

As shown in Figure 1.3, an adversary can masquerade arbitrary domains with a private key that corresponds to a custom root certificate if it is leaked and deceives a limited number of clients that trust the custom root certificate. Note that in one experiment [22], all the eight middleboxes do not remove their custom root certificates from trusted certificate stores at their uninstallation. Furthermore, it is also reported that the passphrase to protect private keys is static in middleboxes. This means that once an adversary can know the secret, he can learn all the private keys from the same products and can conduct a phishing attack to all the clients that trust those keys.

On the other hand, with a shared private key, an adversary can impersonate a limited number of domains listed in a shared certificate (i.e., a `CommonName` and `SubjectAlternativeNames`)

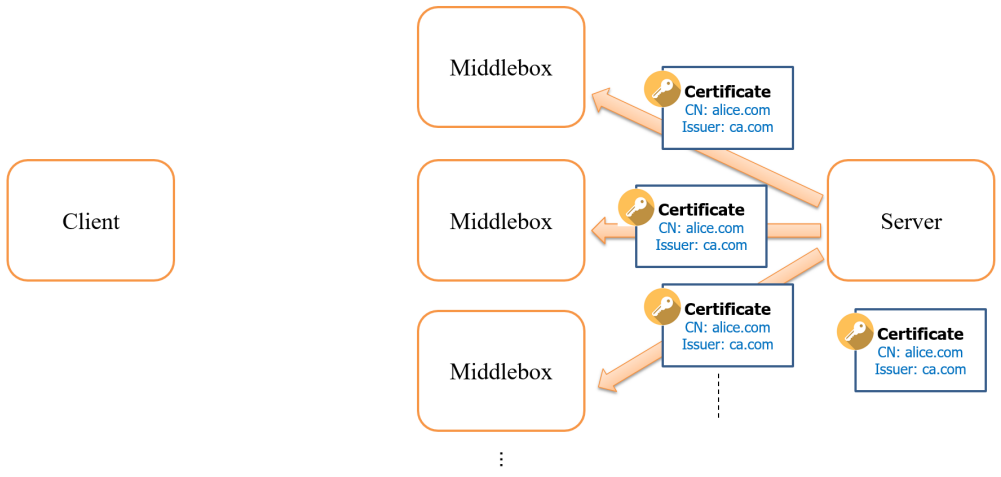


Figure 1.4: **Problems of Key Management in SplitTLS: Private Key Sharing.** Typically, the private key sharing is used in a large infrastructure such as content delivery networks. Therefore, once a private key is uploaded from a server to an infrastructure, it is distributed over the infrastructure, increasing the attack surface.

and cheats arbitrary clients. Note that an infrastructure that utilizes private key sharing is typically large (i.e., many machines included) like CDNs. Therefore, once a private key is shared, it is distributed over the large infrastructure with increasing attack surface of it, as demonstrated in Figure 1.4.

Communication. 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 [22, 25, 101, 18]. 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

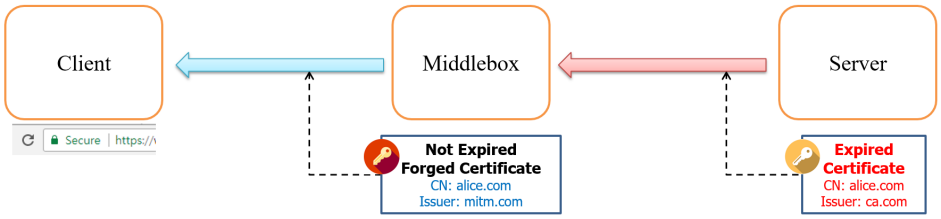


Figure 1.5: **Problems of Communication in SplitTLS: Broken Authentication** Authentication is broken in SplitTLS, as there are some middleboxes that do not validate certificates.

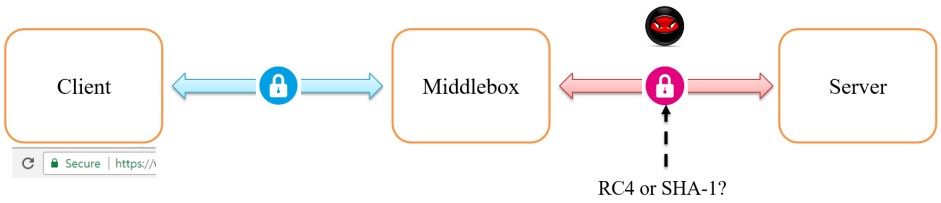


Figure 1.6: **Problems of Communication in SplitTLS: Broken Confidentiality** Confidentiality is broken in SplitTLS, since

showed that some middleboxes do not validate the certificate of the intended server. For example, PrivDog [8] 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) [22, 18].

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

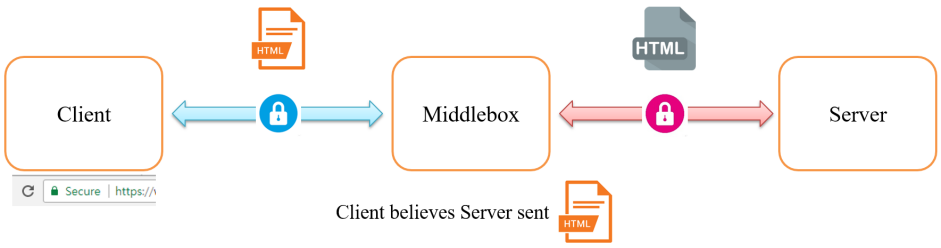


Figure 1.7: **Problems of Communication in SplitTLS: Broken Integrity**

by the proxy [66, 31].

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 [90], 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.

Chapter 2

Literature Review

In this chapter, we survey 23 protocols from 16 research papers and 7 whitepapers. There are 14 protocols for a middlebox-as-a-middlebox out of the 23 protocols, while 9 protocols are relevant to a middlebox-as-an-endpoint. For each type of a middlebox, we categorize protocols into several classes and summarize the lessons from reviewing the protocols, respectively. Unless otherwise stated, we describe the protocols with the scenario of a client that wants to make a middlebox participate in the session for brevity. Note that a client and a server is interchangeable in our description below. We call a finally established secure channel including all the entities as a *session* that consists of *segments* between two entities.

2.1 Middlebox-as-a-Middlebox

2.1.1 Types of Protocols

The 14 protocols for a middlebox-as-a-middlebox can be classified into three types. They are 1) *separate channel approaches* [92, 51, 78], 2) *out-of-band channel approaches* [76, 107, 35, 69, 71, 54, 57], and 3) *in-band channel approaches* [63, 70, 30, 100].

Separate channel approaches

In separate channel approaches, a middlebox performs its functionality on the messages from a client via a separate channel that is secured with a dedicated security scheme such as a

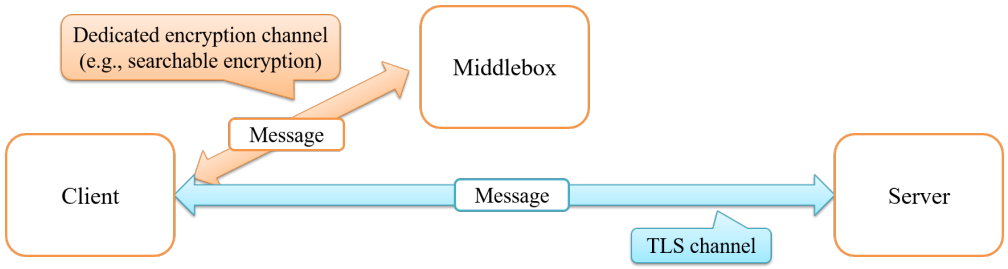


Figure 2.1: **Separate Channel Approaches** The protocols utilize a separate channel toward a middlebox while establishing a TLS session between endpoints. A separate channel utilizes a dedicated security schemes such as searchable encryption to allow least privilege to a middlebox.

searchable encryption. Figure 2.1 demonstrates the high-level concept of these approaches.

The protocols work as follows. As shown in Figure 2.1, a client establishes two channels – one with a server and the other with a middlebox. A channel between endpoints is a standard TLS channel that is established according to the standard TLS protocol, while a channel between a client and a middlebox is a dedicated encryption channel. For example, BlindBox [92] leverages a searchable encryption, Embark [51] uses an order-preserving encryption, and Safebricks [78] utilizes the IPSec [48] with an SGX-based middlebox that is written in Rust [49] for the latter channel. Whenever a client sends/receives a message to/from a server, she first asks a middlebox to inspect a message via a dedicated encryption channel, and she finally sends/receives a message to/from a server via a TLS channel after getting a processed message from a middlebox.

These approaches have the following properties.

First, in some schemes, the privilege of a middlebox is restricted based on dedicated encryption schemes; thus, a middlebox cannot perform excessive functionality (e.g., , modify messages without permission). For example, BlindBox [92] can only perform the keyword matching over a searchable encrypted message and Embark [51] can only conduct IP filtering by leveraging an order-preserving encryption.

Second, dedicated encryption schemes, however, mean that a middlebox cannot provide

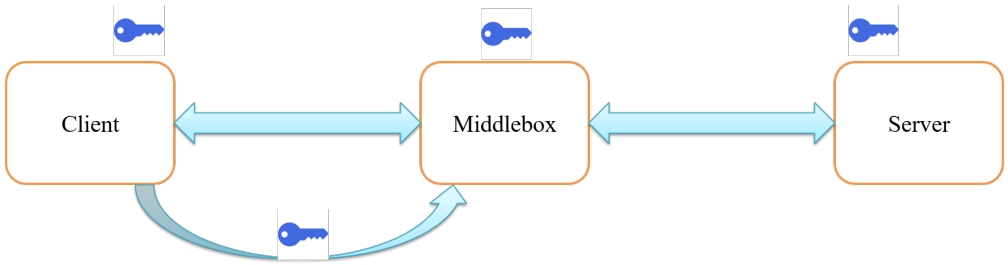


Figure 2.2: **Out-of-band Channel Approaches** After a TLS session is established between endpoints, middleboxes participate in the session by receiving the encryption key from either of endpoints.

a particular functionality until an encryption scheme for it is developed. In other words, a dedicated encryption scheme limits functionality of a middlebox.

Third, these approaches also introduce additional round-trips to a middlebox, which might incur significant networking latency. Note that a client always forwards all the messages to a middlebox before she sends them to or after she receives them from a server.

Fourth, to use these approaches, a new dedicated middlebox should be deployed. For example, a middlebox should support dedicated encryption schemes in BlindBox [92] and Embark [51], while Safebricks [78] require a middlebox implemented with the Rust programming language.

Out-of-band channel approaches

With out-of-band channel approaches shown in Figure 2.2, a middlebox participates in the TLS session by receiving an encryption key from a client via an out-of-band channel.

A naïve approach [76] in this category works as follows. A client establishes a TLS channel with a server and then sends a session key to a middlebox via an out-of-band channel. Then, a middlebox decrypts messages encrypted with a session key, performs its functionality, re-encrypts processed messages, and forwards them to the other side.

We highlight four important aspects regarding a naïve approach.

First, this approach does not require any change on the protocol. What is required is

only a out-of-band channel between a client and a middlebox. Therefore, this approach is *immediately deployable* and does not incur any significant overheads in terms of computation and networking.

Second, without any measure, once a middlebox receives an encryption key, it can read all the encrypted messages and *perform any functionality* on them. In other words, there is no restriction to a middlebox on its behavior. Therefore, a client only sends an encryption key after the key is expired in TLS-RaR [107]. A middlebox just keeps encrypted messages and inspects the messages after it receives the expired key. Although a middlebox cannot look into messages on-the-fly, a client does not need to concern about invalid modifications by a middlebox. SGX-Box [35] utilizes Intel SGX, one of the trusted execution environments, to guarantee the trustworthiness of a middlebox.

Third, a middlebox should support a ciphersuite that is agreed between a client and a server. Otherwise, a middlebox still cannot decrypt encrypted messages. Thus, the TLS KeyShare extension [71] requires a middlebox to advertise its capable ciphersuites or its policy on ciphersuites to both endpoints. In this way, both endpoints can negotiate a ciphersuite supported by a middlebox.

Fourth, this approach can *undermine confidentiality* due to the reused keystream and traffic analysis. To perform functionality, a middlebox should use the same encryption key with the same initialization vector that should not be reused. Therefore, an adversary can exploit the reused keystream to conduct cryptanalysis [71] and can also know that a middlebox modifies a message by comparing an input message with an output message of a middlebox [69]. To address the problem, a client in mbTLS [69] generates different segment keys for all the segments and distributes keys to the corresponding middleboxes (two segment keys per one middlebox).

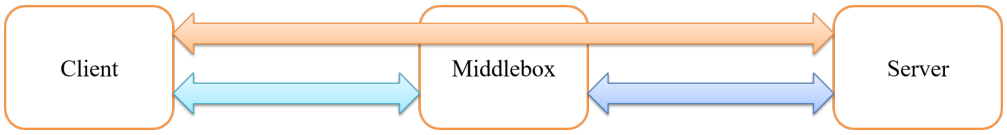


Figure 2.3: **In-band Channel Approaches** Middleboxes intervene in the session by actively participating in the negotiation with endpoints and other middleboxes.

In-band channel approaches

In in-band channel approaches, a middlebox actively contribute to establishing keys (by generating key materials) and finally intervenes into the session. The concept is demonstrated in Figure 2.3.

The protocols are executed as follows. A client initiates a TLS extension with a server and middleboxes. Each entitiy has its role and exchanges key materials to establish keys according to the protocol. Finally, middleboxes have their own keys such as segment encryption keys or message authentication code (MAC) keys, and exchange messages utilizing the keys.

In the TLS ProxyInfo extension [63], for instance, endpoints and all the middleboxes exchange their key materials with adjacent entities, and establishes an encryption key per segment. mcTLS [70], EFGH [30], and TLMSP [100] perform the Diffie-Hellman (DH) key exchange between an endpoint and all the middleboxes.

We find three important features of in-band channel approaches.

First, a middlebox explicitly authenticates itself to entities to participate in the key generation process. All the studies in this type assumes that a middlebox has its own certificate and a client authenticate all or parts of the middleboxes in the session.

Second, a server can perform cryptographic operations to support a client to understand a middlebox. For example, a server and a client agree on a list of middleboxes authorized to be involved in the session in mcTLS. Also, a server and a client contain an endpoint MAC in the record layer to show whether a middlebox modifies a message or not. Furthermore, TLMSP introduces an audit trail that logs a series of hash values of a message passing through

middleboxes.

Third, since a middlebox actively participates in the key establishment, a middlebox should generate and verify signatures, which incurs computational overheads. For example, a middlebox should generate a signature over a key material and should verify a signature from other entities in mcTLS, which incurs additional computation overheads compared with other types of approaches.

2.1.2 Takeaways

We have learned the following lessons from reviewing the 14 protocols for a middlebox-as-a-middlebox.

First, we should consider the *least privilege of a middlebox* with a protocol. Many past studies including mcTLS [70], BlindBox [92], Embark [51], Safebricks [78] provide their own method to limit a middlebox not to perform functionality with excessive permission.

Second, a server can help a client to understand a middlebox. As seen in in-band approaches, a server can generate an endpoint MAC or a signature of a content; thus, a client can understand how a message passes through middleboxes.

2.2 Middlebox-as-an-Endpoint

2.2.1 Types of Protocols

The 9 protocols can be categorized into 4 types, which are 1) *certificate-based approaches* [56], 2) *redirect-based approaches* [95, 26, 53], 3) *key server-based approaches* [98, 13, 12], and 4) *token-based approaches* [9]

Certificate-based approaches

As shown in Figure 2.4, there are two proposals of certificate-based approaches called the DANE-based approach and the Name Constraint Certificate approach, both of which are proposed by J. Liang *et al* [56]. Those utilize existing infrastructures such as the DNS-based

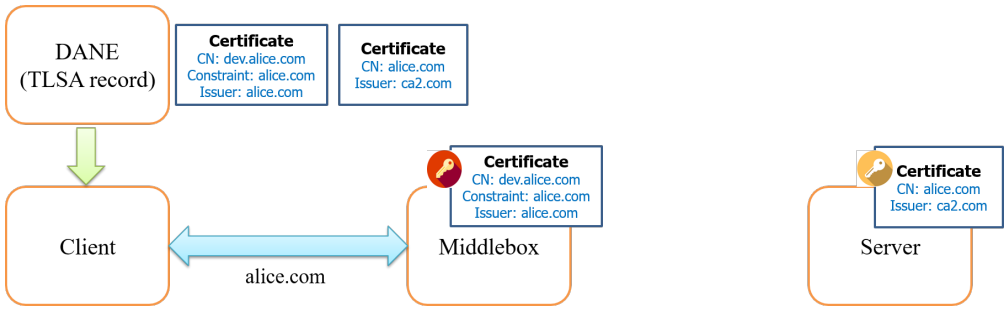


Figure 2.4: **Certificate-based Approaches** This approach leverages the existing infrastructure such as PKI or DANE to authenticate and authorize middleboxes.

Authentication of Named Entities (DANE) [36] and the Public Key Infrastructure [38]. In the DANE-based approach, a server adds his certificate as well as a middlebox's certificate to the TLSA resource records in his zone file. On the other hand, a server generates a key pair including a name constraint certificate and delegates them to a middlebox in the Name Constraint Certificate approach. The middlebox that receives a name constraint certificate can use it to authenticate herself as a valid delegator from a server.

We state two important points below.

First, these approaches are *deployable* in that both of them leverage the existing infrastructures. There is no need to upgrade a server to deploy these schemes, while minimal changes are required for a client (e.g., parsing TLSA resource records or validating name constraint certificates).

Second, these approaches, however, incur additional efforts to a server since it requires a server to manage keys. For instance, a server should configure his TLSA resource records and should manage keys as an intermediate certificate authority.

Redirection-based approaches

In the redirection-based approaches, a client first connects to a server and is redirected to a middlebox; then, the final session is established between a client and a middlebox, as in Figure 2.5.

Figure 2.5: **Redirection-based Approaches** A middlebox is forwarded from a server; therefore, a client can explicitly authenticate a middlebox that is authorized by a server by a redirection.

A naïve approach for this type of protocols is the HTTP 302 Redirection protocol [95]. Initially, a client executes a TLS handshake with a server and sends an HTTP request to a server. A server responds with an HTTP 302 status code that performs URL redirection toward a middlebox. Then, a client connects to a middlebox with executing a TLS handshake and sending an HTTP request. Finally, a middlebox repsonds to the request. Since the HTTP 302 Redirection protocol requires two TLS handshake, meaning that the elapsed time required to establish a session with a middlebox is doubled compared with the current practice. To reduce such latency, BlindCache [26] uploads content encrypted with a previously generated encryption key on a middlebox in advance and redirects a client toward a middlebox via HTTP (not HTTPS) while giving an encryption key by leveraging the HTTP out-of-band field [80]. A client gets not only an URL of a middlebox, but also the signatures of contents from a server in Stickler [53]; thus, a client can verify the integrity of content on a named middlebox.

We can get two learnings from this type of approaches.

First, a server can provide a client with a *necessary information* about a middlebox. A server presents a client with a name of a middlebox by a URL, a client can authenticate a middlebox with a middlebox's certificate. Furthermore, as described above, Stickler offers content signatures to a client to guarantee the integrity of the content. This is possible since a client directly connects to a server at least once.

Second, these approaches, however, require a round-trip to a server, which incurs *sig-*

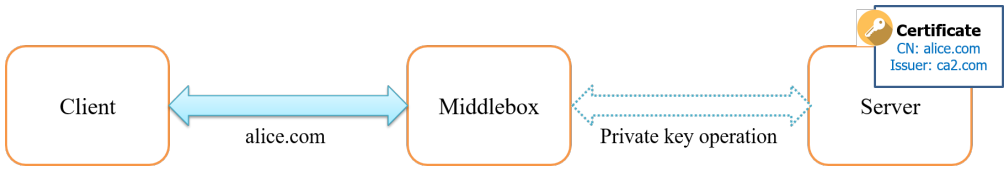


Figure 2.6: **Key Server-based Approaches** A server performs a private key operation on request from a middlebox; thus, a server does not need to delegate a private key to a middlebox.

nificant networking latency. In general, a middlebox-as-an-endpoint is deployed to reduce networking latency by locating a middlebox near a client. Unfortunately, these approaches reduce the merits of a middlebox-as-an-endpoint.

Key server-based approaches

In key server-based approaches, a server deploys a key server that operates a private key operation such as a signature generation or an asymmetric decryption on request from a middlebox, which are described in Figure 2.6.

The protocols are executed as follows. A client initiates a TLS handshake with a middlebox. During a handshake, a middlebox requests a private key operation to a key server to generate `ServerKeyExchange` in TLS 1.2 or `CertificateVerify` in TLS 1.3, which includes a signature of a server. The session between a middlebox and a key server is a TLS session that is established with mutual authentication to provide a private key operation only to an allowed middlebox. The TLS session is finally established between a client and a middlebox, whlie a client believes that she is communicating with a server.

Note that this approach is currently serviced by Cloudflare as $\mathsf{Keyless-SSL}$ [98], which is standardized as LURK [20, 21]. $\mathsf{Keyless-SSL}$ is also formally treated and is improved as $\mathsf{3(S)ACCE-K-SSL}$ [13]. Instead of a private key operation, a middlebox requests a session key to a server while providing a server with an attestation report in SPX.

We find the following three features of these approaches.

First, *no private key sharing* is needed; thus, a middlebox is only required to manage its

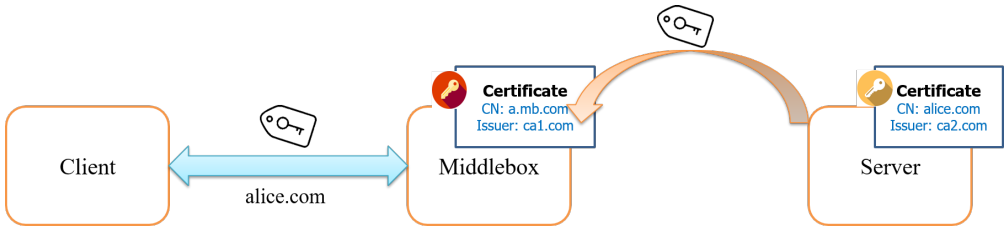


Figure 2.7: **Token-based Approaches** A server previously issues an authorization token to a middlebox. Whenever a middlebox receives a request from a client, it uses the token to prove its authority granted by a server.

own key that is used to authenticate itself to a key server. The number of secrets is dramatically reduced compared with the current CDN practice; thus, a server does not need to concern about his private key being leaked on a middlebox.

Second, the protocol is compatible with a current client; thus, a client does not need to modify his software. Only a server and a middlebox should upgrade their software to support the new protocol.

Third, as redirection-based approaches do, these approaches also adds a round-trip to a (key) server, which incurs significant networking delay.

Token-based approaches

In the token-based approaches, a middlebox utilizes a token issued by a server in advance to authenticate itself as a server or a valid delegator, as shown in Figure 2.7.

The example of these approaches is DelegatedCredential that is currently being evaluated by Cloudflare [34], Facebook [99], and Mozila [42] and is executed as follows.

A server generates a short-term keypair and signs it with a server's long-term private key. The resultant signature is called a delegated credential. Then, a server delivers a short-term keypair and a delegated credential to a middlebox with his certificate. Whenever a client initiates a TLS handshake, a middlebox responds with a server's certificate as well as a delegated credential. A client finally confirms that a middlebox is authorized by a server.

There are three points that we note regarding these approaches.

First, unlike previous two approaches – redirection-based and key server-based approaches – these approaches do not need an additional round-trip to a server. A token can be generated and delivered to a middlebox in advance; thus, the token issuing procedure is independent of a communication. Therefore, the networking delay is nearly similar to the current CDN practice.

Second, to support the new protocol, all the entities should upgrade their software. A server and a middlebox should deploy a token generating server/client, while a client and a middlebox should upgrade the TLS library to support a TLS extension that utilizes a token.

2.2.2 Takeaways

We describe what lessons we have learned from our literature review on protocols for a middlebox-as-an-endpoint.

First, the number of secrets should be minimal. If a middlebox should have many private keys to perform its functionality, a middlebox can be a target of an adversary. Note that a middlebox in KeylessSSL or HTTP 302 Redirection only has its private key (not others). On the other hand, DelegatedCredential

Second, the key management overhead should not be placed on a server. A server is required to manage keys (e.g., key revocation management) in Name Constraint Certificate. It is known that the key management requires heavy overheads; thus, it is undesirable to place the burden on a server.

Third, additional round-trips to a server should be avoided. Many protocols show that additional round-trips significantly incurs networking latency. Considering a middlebox-as-an-endpoint is deployed to enhance the performance, additional round-trips should be removed.

Chapter 3

MATLS: How to Make TLS middlebox-aware?

3.1 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 [91].

However, the practice of using middleboxes is not compatible with Transport Layer Security (TLS) [23, 82] — 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) [81] becomes increasingly common (more than 50% of total HTTP traffic is now encrypted by TLS [29, 68]), 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 [45], 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 [22, 25, 103, 73, 101]; 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 [22, 25, 103]. Similarly, a middlebox that splits a TLS session may support only weak ciphersuites, which are vulnerable to known attacks such as the Logjam attack [3] or the FREAK attack [10]. Even worse, it has been reported that middleboxes are being used to inject malicious code [101, 73, 18]; for example, Giorgos et al. [101] 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 [92, 51, 78, 35, 60, 71, 70]. These studies can be largely classified into three main categories: encryption-based, trusted execution environment (TEE)-based, and TLS extension-based. First, BlindBox [92] and Embark [51] proposed to use special encryption schemes such as order-preserving encryption to allow

[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.

middleboxes to perform their functionality over encrypted packets. Second, SafeBricks [78] and SGX-Box [35] leveraged TEEs such as Intel SGX to make middleboxes trustworthy. Third, several studies sought to extend the TLS protocol [70, 60, 71, 63, 69] 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 [60] 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 [70], TLMSP[2], and TLS Keyshare extension[3] [71] 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 [63]. 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 [70], the

[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.

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 [64], 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 chapter is organized as follows. First, we explain our trust and threat model (§3.2). Then, we describe how to make middleboxes auditable (§3.3), and design the MATLS protocol (§3.4). We verify our security model (§3.5), evaluate the performance overhead of MATLS (§3.6) and present our concluding remarks (§3.8).

3.2 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 misconfigured or incorrectly implemented, they may accept invalid certificates, use deprecated ciphersuites, or attempt to inject unwanted or malicious content [101, 73].

(4) *Certificate Authority (CA)*: An organization that issues and revokes certificates. A CA issues a certificate to 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 [52]) 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 §3.3. Alternatively, the CT system can be assumed to accommodate middlebox certificates as well.

Adversary capabilities. We accept the Dolev-Yao model [24] 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. .

3.3 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.

3.3.1 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, `MiddleboxInfoAccess` where its ASN.1 syntax is defined as follows.

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

MiddleboxDescription::=  SEQUENCE {
  MiddleboxInfoType     OBJECT IDENTIFIER,
  MiddleboxInfo         GeneralName}
```

  For example, *permission* can be one of the `MiddleboxInfoType` 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 `MiddleboxDescription`.

3.3.2 Middlebox Transparency

We introduce an MT log server that publicly records middlebox certificates. The operation of MT is similar to that of CT [52]. 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.

3.3.3 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 §3.4, 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` [41], 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 [38] or OCSP [67]. 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.

3.4 Middlebox-aware TLS (MATLS)

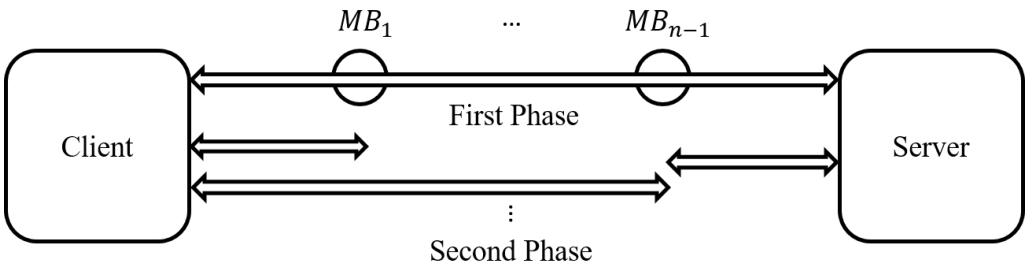
In this section, we describe the MATLS protocol, which is designed to allow middleboxes to participate in a TLS session. As we have middleboxes equipped with certificates, we extend the security goals of TLS to the seven objectives below, divided into three categories. For the sake of exposition, we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

3.4.1 Security Goals

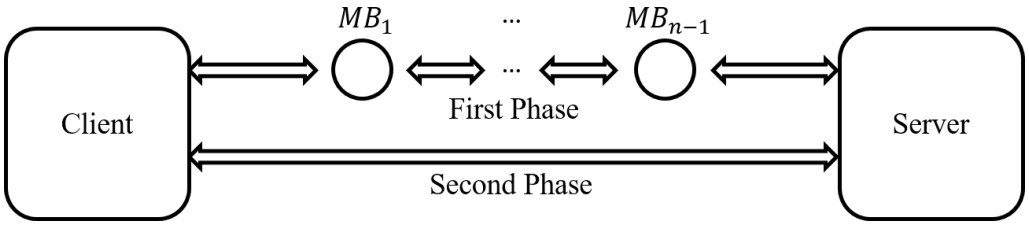
Authentication: . Similar to the authentication process of TLS certificates, clients should be able to receive and check the validity of the certificate of the server that the clients intended to connect. This should hold even when there are middleboxes splitting the TLS connection between them. Thus, 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*



(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.

Figure 3.1: **Two approaches to establish a TLS session with middleboxes.** We adopt the bottom-up approach since it efficiently supports incremental deployment.

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*.

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_i	Message sent from e_i
$a\|\|b$	a concatenated with b
$PRF(a, b, c)$	Pseudorandom function in [23] 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_i, pk_i)	Entity e_i's (secret key, public key) pair
$Cert_i$	Entity e_i's certificate
ID_i	Identity of e_i. $ID_i = H(pk_i)$
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
$ak_{i,j}$	Accountability key of e_i established with e_j (We simply write ak_i when j is fixed in the context)
$HMAC_i$	The result of $Hmac(k, m)$ by e_i
ML_i	Modification log generated by e_i

Table 3.1: Notation used in the description of MATLS

3.4.2 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 3.1. 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 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 [70], TLMSP, and TLS Keyshare extension [71], 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 [69]. 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 3.2, alongside their notation in Table 3.1.

Explicit Authentication	
Proof	A sequence of certificate blocks, including the server certificate and any middlebox certificates with their signed certificate timestamps
Description	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 [52] and DANE [36]
Security Parameter Verification	

Proof	Security parameters of every MATLS segment including a negotiated TLS version, an agreed ciphersuite, and a transcript of the handshake
Description	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	
Proof	A modification log that keeps track of the modifications of a packet
Description	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 3.2: **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.

(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, 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 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:

$$ID_i||p_{i,i+1}||Sign(sk_i, Hmac(ak_{i,0}, p_{i-1,i}||p_{i,i+1}))$$

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 $ID_n||Sign(sk_n, Hmac(ak_{n,0}, p_{n-1,n}))$ 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 [79].

(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_i), (2) a digest of the received message ($H(m_{i+1})$), and its identifier (ID_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_i:

$$ID_i||H(m_{i+1})||Hmac(ak_{i,0}, H(m_i)||H(m_{i+1}))||ML_{i+1}$$

Here, we can apply some optimization techniques to reduce the size of the MLs in specific scenarios. First, the server does not have a prior message, thus the ML_n can be defined as $ID_n||Hmac(ak_{n,0}, H(m_n))$. 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_i by (1) simply generating a $HMAC_i$ from the previous $HMAC_{i+1}$ and (2) omitting its received digest ($H(m_{i+1})$) and even its ID (ID_i). Thus, if the client detects a omitted ID while parsing the received ML, it can assume that the message has not been modified among the middleboxes with the omitted IDs. For example, if an entity (e_i) receives a message that has never been modified, the ML

that the entity received will be

$$ID_n||Hmac(ak_{i+1,0}, Hmac(ak_{i+2,0}, \cdots, Hmac(ak_{n,0}, H(m_n))).$$

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

$$ID_i||H(m_{i+1})||Hmac(ak_{i,0}, H(m_i)||H(m_{i+1}))||ID_n||HMAC_{i+1},$$

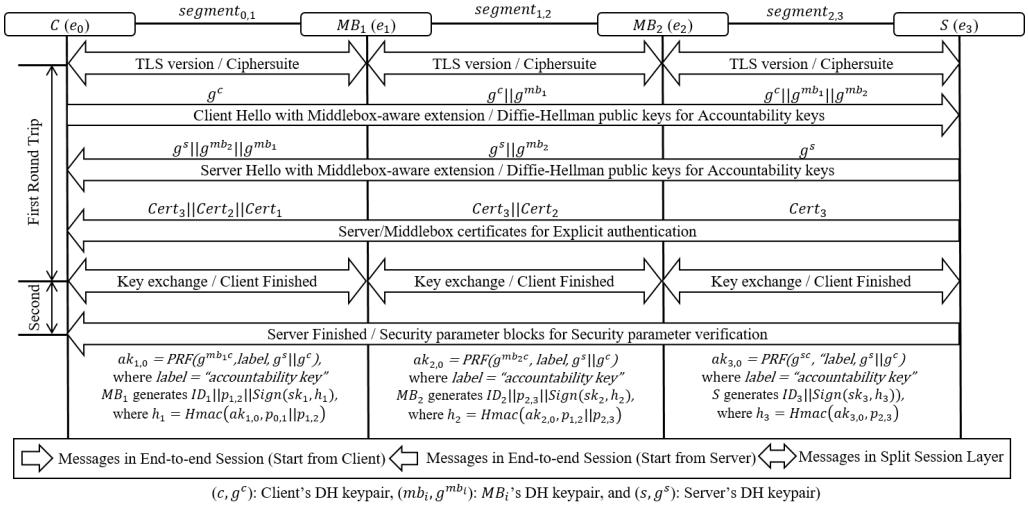
which implies that the message between the middlebox e_{i+1} and e_n has never been modified.

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.

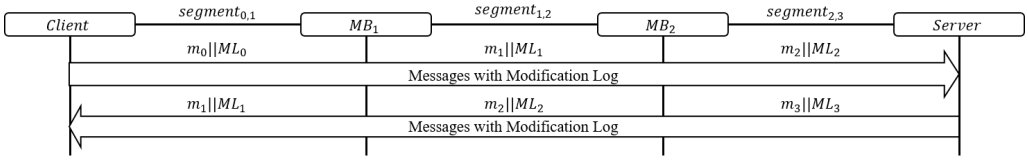
Note that this optimization has a limitation if a write middlebox lies. Let say two consecutive middleboxes simply perform HMAC over the received ML as if they are read-only middleboxes, but the message is modified. In this case, a client cannot identify the lier; thus, a level of accountability is decreased. Nevertheless, we propose the optimization to reduce an amount of the hash values included in the record messages.

3.4.3 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 3.2a. In the first round-trip, the client expresses its preference to perform the MATLS protocol by adding the `MiddleboxAware` 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. This process is repeated at every middlebox on the way to the server.



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



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

Figure 3.2: **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.

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 [23] 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.

3.4.4 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 3.2b. 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.

3.5 Security Verification

We analyzed the security goals of the MATLS protocol using Tamarin [64], 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 $\texttt{l} \rightarrow \texttt{[a]} \rightarrow \texttt{r}$. 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.

3.5.1 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.

3.5.2 Adversarial Model

As described above, we adopt a Dolev-Yao adversary that can be modeled with several rules as follows:

$$\texttt{Inject} := [\ !\mathrm{K}(x)\ ] \mapsto [\ \mathrm{Net}(x)\ ]$$

$$\texttt{Block} := [\ \mathrm{Net}(x)\ ] \xrightarrow{\mathrm{K}(x)} [\ !\mathrm{K}(x)\ ]$$

$$\texttt{Adv\_Pub} := [\ -\ ] \xrightarrow{\mathrm{K}(A)} [\ !\mathrm{K}(A : pub)\ ]$$

$$\texttt{Adv\_Fr} := [\ \mathrm{Fr}(x)\ ] \xrightarrow{\mathrm{K}(x)} [\ !\mathrm{K}(x)\ ]$$

$$\texttt{Fun}_\texttt{f} := \left[\begin{array}{l} !\mathrm{K}(x_1) \\ !\mathrm{K}(x_2) \end{array}\right] \xrightarrow{\mathrm{K}(f(x_1,x_2))} [\ !\mathrm{K}(f(x_1,x_2))\ ]$$

$$\texttt{Corrupt\_Ltk} := [\ !\mathrm{Ltk}(A,k)\ ] \xrightarrow{\mathrm{Corrupt}(A)} [\ !\mathrm{K}(k)\ ]$$

$$\texttt{Corrupt\_L} := [\ !\mathrm{ShKey}(A,B,k)\ ] \xrightarrow{\mathrm{Corrupt}(A)} [\ !\mathrm{K}(k)\ ]$$

$$\texttt{Corrupt\_R} := [\ !\mathrm{ShKey}(A,B,k)\ ] \xrightarrow{\mathrm{Corrupt}(B)} [\ !\mathrm{K}(k)\ ]$$

Figure 3.3: **A Dolev-Yao adversary in Tamarin.** A Dolev-Yao adversary can be modeled with several rules such as Inject,Block, and Corrupt in Tamarin.

3.5.3 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 3.5.3. 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).

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 3.3: **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.

The results of the analysis show that the MATLS protocol satisfies the core security goals. The full Tamarin implementation can be found at our public repository at `https://github.com/middlebox-aware-tls/matls-tamarin`.

3.6 Evaluation

3.6.1 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)[4]. The server-side middlebox and the server are equipped with an Intel Xeon CPU E5-2676 at 2.40GHz with 1GB memory. We used a virtual machine

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

with an Intel Core i7 at 2.30GHz and 1GB memory for the client-side middlebox, and a virtual machine with an Intel Broadwell CPU at 3.30GHz and 1GB memory 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 3.4.

After establishing an MATLS session, the client requests an HTML page of 1KB with an HTTP GET message, respectively, terminating the connection after completing the download of the corresponding HTTP response. Each plotted value is the average of 100 measurements. We compare the performance overhead of MATLS with those of SplitTLS and mcTLS [70], 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.

Testbed	C-MB_C	MB_C-MB_S	$MB_S - S$
Intra-country	1.136ms	4.944ms	0.551ms
Intra-region	1.136ms	35.896ms	0.537ms
Inter-region	1.136ms	192.818ms	0.610ms

Table 3.4: **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.

3.6.2 HTTPS Page Load Time

We first evaluate the time elapsed to fetch an 1KB file from the server in the MATLS protocol, which is compared with the SplitTLS and mcTLS protocols.

Figure 3.4 summarizes the time taken from starting a TCP handshake to finishing the download of the content. We observe that the MATLS protocol introduces a slight delay

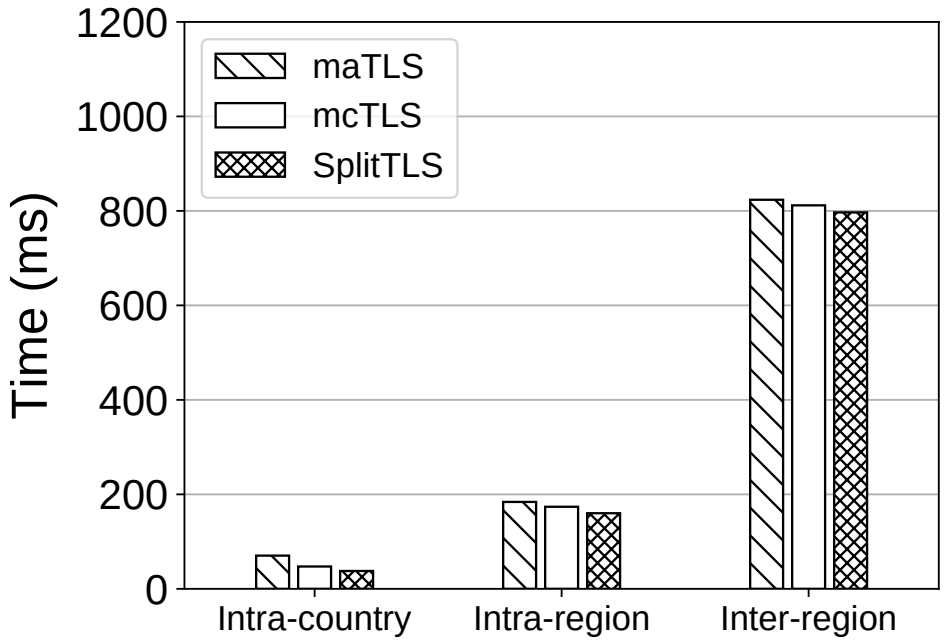


Figure 3.4: **HTTP Load Time**

(10.22ms – 32.52ms) compared to SplitTLS and mcTLS 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.

To quantify the overhead that the MATLS record protocol requires, Figure 3.5 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 delay time of the MATLS record protocol is similar to that of the SplitTLS and mcTLS

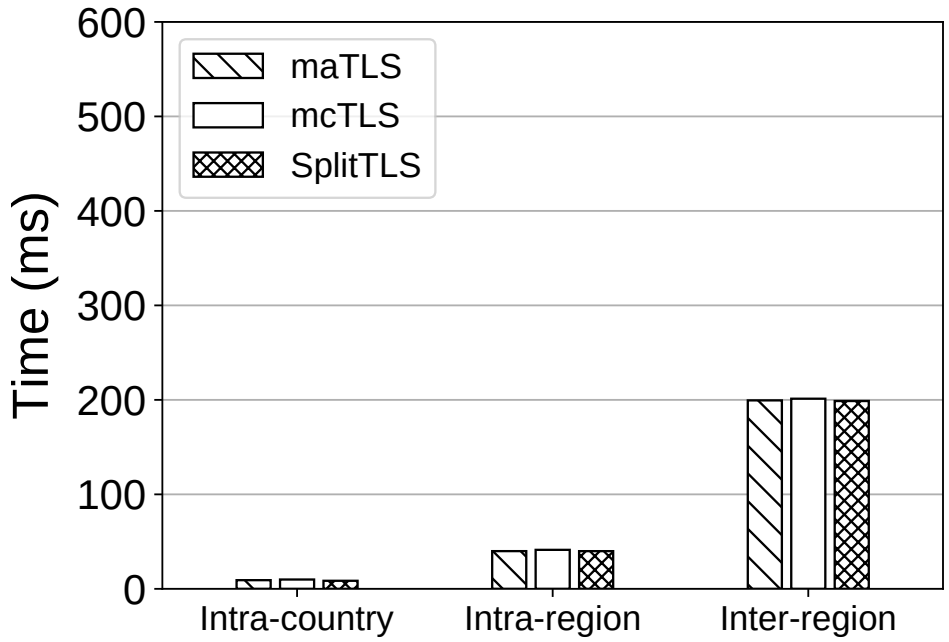


Figure 3.5: **Data Transfer Time**

record protocols. For example, in the intra-region testbed scenario, the data transfer time is 39.92ms, 39.90ms, and 41.28ms in MATLS, SplitTLS, and mcTLS, respectively.

From Figure 3.4 and Figure 3.5, 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.*

3.6.3 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

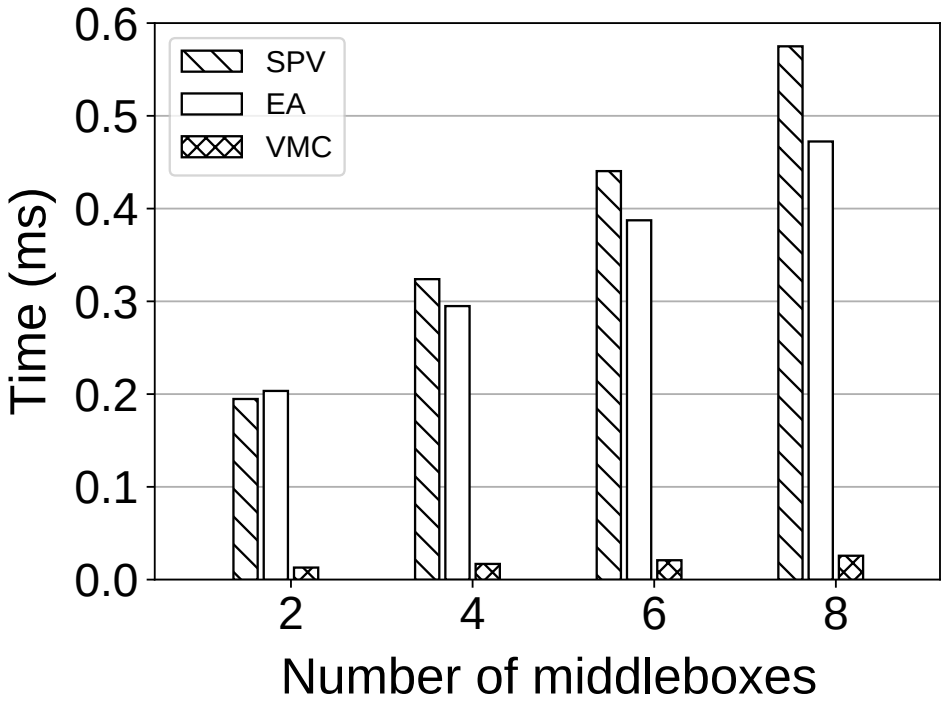


Figure 3.6: **Integrity Verification Time**

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 3.6).

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

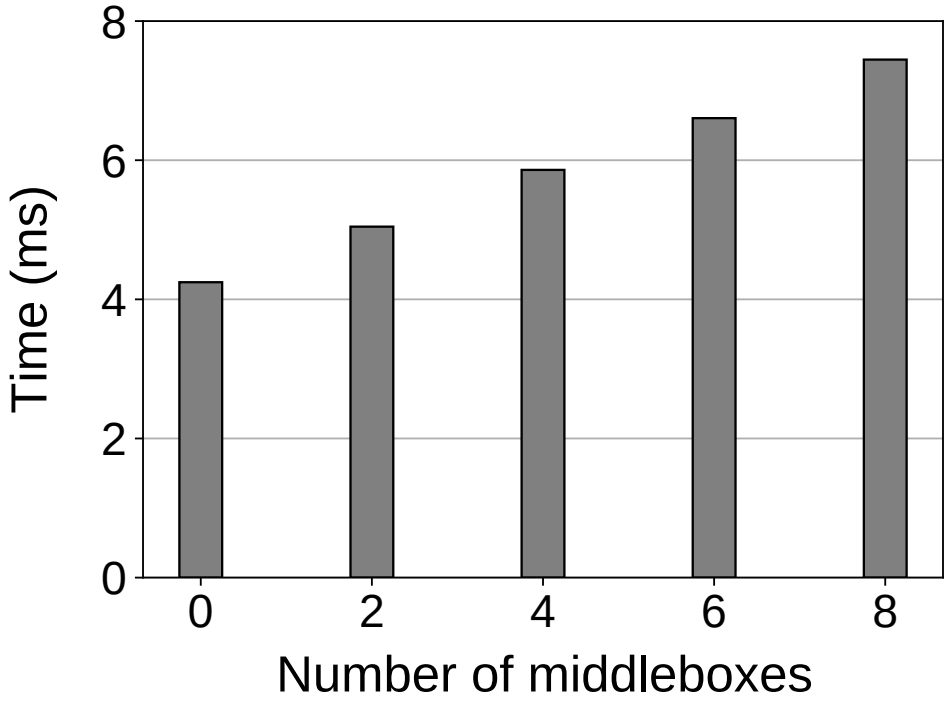


Figure 3.7: **CPU Processing Time**

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.

3.6.4 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 3.7, 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.

3.7 Discussions

3.7.1 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.

3.7.2 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.

3.7.3 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.

3.7.4 TLS 1.3 Compatibility

TLS 1.3 [82] 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.

3.7.5 Mobility Support

Since mobile devices are pervasive, we should consider how MATLS responds to a mobile environment. We follow the terms from [62]. A *mobile node* is a node that can change its point of attachment to the network without breaking open sessions. A mobile node can be either of a mobile host or a mobile router, the former of which can forward packets and the latter of

which cannot.

We would like to highlight two points regarding mobility support of MATLS.

First, MATLS does not need to deal with a middlebox located on public servers that have their own IP addresses. Even if a mobile node changes its IP address, the TCP session is maintained between a mobile node and a middlebox according to a mobility support mechanism in a mobile network.

Second, a mobile host should re-establish an MATLS session when a middlebox is located on a gateway or a base station. This is because if a mobile host moves to another network, the session between a mobile host and a middlebox is disconnected.

3.7.6 P2P Communication

The recent video streaming service and web conferencing commonly use the WebRTC technology that a central server only serves as a broker to establish a session between two peers. The MATLS protocol is also compatible with the p2p communication. After two peers get information about each other with the help of a broking server, two peers can simply execute the MATLS protocol to make a middlebox participate into the p2p session.

3.8 Conclusion

In this chapter, 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, security parameter blocks, and modification logs 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 33ms, 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.

Chapter 4

TLS-SEED: How to SEcurely Communicate with EDge Computing Platforms?

4.1 Introduction

Due to the explosive growth of Internet-connected devices and the popularity of cloud-based services, cloud platforms suffer from a scarcity of bandwidth [104]. To mitigate these problems, *edge computing* (a.k.a., fog computing) [59, 93, 86, 14] has emerged to support fast delivery and improved bandwidth utilization, by *placing computing resources at the* edge of a network.

Both industry [74, 97] and the research community [11, 58] have designed their edge computing platforms, but based on a similar architecture. Application service providers (ASPs) move their applications (called edge applications) to devices in the network (called edge devices), which are typically managed by third-party platform providers (e.g., Intel [40]). As application service providers can push their applications to multiple edge devices as a form of container images without designing and deploying their platforms, edge computing has attracted many application service providers in the area of content caching [4, 61, 104, 105], augmented reality services [89], and vehicle control systems [85].

However, edge computing has fundamental security challenges, mainly due to two factors:

edge applications run outside the realm of the application service providers, and multiple edge applications share a physical edge platform, thus exposing themselves to increased threats from tampering or tapping compared with machines in private data centers [7, 12].

For this reason, several edge computing platform specifications [84, 33] recommend using Transport Layer Security [23, 82] between clients and edge applications by default, and conducting attestation by leveraging hardware security such as Trusted Platform Module (TPM) to allow clients to attest the integrity of the edge application. However, there are still unsolved challenges to achieve secure communication with edge computing platforms, which we explain in the following:

Application service provider: risky private key management. As an edge application on an edge computing platform communicates with clients on behalf of an application service providers, the application service provider needs to embed its *private key* into the edge application running on the edge platform, which is *not* managed by the application service provider. Worse yet, an edge computing platform (1) hosts many edge applications, thus inevitably maintains multiple private keys, and (2) has to be distributed over multiple places administered by multiple network operators. Thus, edge computing requires more sophisticated techniques to deal with such a broad attack surface.

Client: inefficient remote attestation. Since edge applications may process sensitive information of clients, clients want to attest the integrity of the edge application which they are communicating with. There have been a few studies enabling remote attestation for clients [39, 69], but there are still two main challenges.

First, as the client may want to quickly check the validity of the edge application (e.g., fast response from the augmented reality application), the remote attestation should not impose a substantial performance overhead on the client; however, prior studies [39, 69] do not consider this issue and hence incur extra round-trips to attestation services (e.g., Intel attestation service), increasing the network overhead.

Second, to verify a software attestation during remote attestation, a client should know

the trusted hash value of an edge application in advance. However, she might not know whether she is communicating with an edge application or the application service provider's server, since a client may transparently connect to an edge application. Thus, she might need other round-trips or other measures to know a trusted hash value for attestation, which incurs performance degradation.

In this chapter, we propose SEED, which provides trustworthy edge computing. SEED addresses the two challenges while providing additional benefits: *accountability* to application service providers and *visibility* to clients. This is done by combining the TLS protocol and remote attestation from Trusted Execution Environment (TEE) technologies (i.e., both Intel SGX and ARM TrustZone), without sharing any long-term private keys with a third-party, which we call TLS-SEED. In particular, we introduce a CROSS CREDENTIAL (CC), a short-term data structure to show a *trust relationship* between an application service provider and an edge device in the SEED platform to a client[1]. The CC also proves the integrity of the edge application from the application service provider. We extend TLS to use a CC, which is dubbed TLS-SEED. TLS-SEED consists of two TLS extensions (collectively called TLS-SEED) – a TLS-BACKEND-SEED protocol (an application service provider's issuing a CC to an edge device) and a TLS-FRONTEND-SEED protocol (a client's verifying a CC presented by an edge device).

With a CC, an application service provider can (i) deploy his edge application without embedding his private key, and (ii) achieve *accountability* by issuing a CC after attesting his application. A client can (iii) get *visibility* of an edge computing platform (i.e., the communicating counterpart is a trusted computing platform), by efficiently performing remote attestation of an edge application.

We analyze the security and evaluate the performance overhead of TLS-FRONTEND-SEED since this protocol allows a client and a SEED device to exchange application data with delay-sensitive requirements. To this end, we introduce the ACCE-SEED model by extending

[1] An edge device in the SEED platform is also called a SEED device.

the ACCE model [43] that is generally used to prove the standard TLS and prove TLS-FRONTEND-SEED is ACCE-SEED-secure. Then, we implement TLS-FRONTEND-SEED with OpenSSL (over both Intel SGX and ARM TrustZone) and compare it with other schemes in terms of the latency and CPU overhead.

In summary, we make the following contributions:

- We design a CROSS CREDENTIAL (CC) that enables an application service provider to deploy his edge application without sharing his private key and a client to attest the application efficiently.

- We propose the TLS-SEED protocol to incorporate CCs and formally prove the TLS-FRONTEND-SEED protocol by leveraging the ACCE-SEED model.

- We demonstrate the feasibility of SEED by implementing TLS-SEED with OpenSSL over Intel SGX and ARM TrustZone.

The remainder of this paper is organized as follows. We introduce the the background with related work (§4.2). Next, the high-level overview of the SEED platform is described (§4.3). We then detail the SEED platform with a focus on a CC and TLS-SEED (§4.4), followed by a security analysis leveraging the ACCE-SEED model (§4.5) and a performance evaluation of TLS-SEED (§4.6). We finalize with concluding remarks (§4.8).

4.2 Preliminary

In this section, we first review edge computing platforms and the networking patterns of the edge applications. We then describe trusted execution environments (TEE) such as Intel SGX and ARM TrustZone. Finally, we briefly explain about TLS variants that aim to avoid private key sharing or to combine remote attestation.

4.2.1 Edge Computing

Edge computing is an emerging paradigm that brings computation and data storage closer to end-users. New applications such as autonomous vehicles or augmented reality require delay-sensitive services; thus, placing computing resources near clients to reduce delays has become a viable solution.

Edge computing platforms. There have been many studies and trials for edge computing platforms. For instance, industry stakeholders have attempted to standardize ETSI Multi-access Edge Computing [74]. Companies and academic institutions joined the OpenFog consortium for similar standardization [33]. Academic groups also propose Cloudlet [87], AirBox [11], and Paradrop [58]. We consider a general edge platform from the ones in the literature, as explained below.

Application service providers (ASPs) develop their edge applications, which are deployed and run on central clouds or edge clouds leveraging virtualization technologies such as virtual machines or containers. Developed applications are typically registered with an image registry in the edge computing platform. Then, edge devices in the platform can load the edge applications from the registry if needed. We assume that the ASPs are not involved in the registration/migration process. That is, they need not consider how to distribute/replicate their edge applications.

Networking of edge applications. In the edge computing platform scenario under consideration, a client application (on her device) directly communicates with an edge application running on an edge device as described in [74, A.12]. To establish a session between a client and her nearest edge application, the platform utilizes DNS redirection or a TCP/IP termination based on the domain name or the destination address of the intended server. Then, the platform redirects packets from a client to an edge application and vice versa. Thus, a client communicates with the intended server without knowing whether she connects to an edge device (running an edge application), or to a central cloud server.

Due to the above interception, there are two channels called *frontend* and *backend* channels in edge computing. The former is established between a client and an edge application and is used to provide delay-sensitive services that require fast responses (e.g., edge caching, augmented reality, and autonomous vehicle). The latter is established between the edge application and a cloud server, the channel is used only when an edge application needs any data from a cloud server. For example, an edge caching application gets content from a cloud server only in case of cache misses. An IoT aggregator, after receiving data from client devices, intermittently reports aggregated data to a cloud server. This indicates that when a client establishes a session with an edge device, a cloud server is participating in the session typically in an intermittent and on-demand fashion. Also, a backend channel can be used separately from a frontend channel and delay-tolerant.

Security in edge computing platforms. Security in edge computing is crucial due to the following reasons. First, edge computing platforms can be a third party with a massive infrastructure in the sense that the ASP has no control over the platforms. Another essential nature is that edge devices are often vulnerable to attacks compared with machines in the data center [7]. For example, WiFi access points or home routers are widely used edge devices in the edge computing platform [11, 58, 7]. Thus, there can be several physical attacks, side-channel attacks, or other software-based attacks against the edge computing platform. Therefore, essential resources such as private keys or sensitive private data of edge applications should be protected, and the integrity of the platform should be guaranteed.

Due to the above reasons, several proposals present guidelines such as using the (D)TLS protocol for network security [84], leveraging Hardware Security Module (HSM) for secure storage [84], or performing attestation with Trusted Platform Module (TPM) [33]. However, the current standardizations of edge computing have not answered correctly for (i) how to securely distribute (or manage) private keys to edge applications for TLS, or (ii) how to perform remote attestation. There are only some guidelines but no systematic mechanisms. These issues are left as developers' responsibilities [84] or unspecified yet [33].

4.2.2 Trusted Execution Environments

Trusted execution environment (TEE). A TEE is a secure area within a main processor and a memory. It provides an isolated execution region to run a program securely from other untrusted software components such as an operating system as well as applications. The TEE and other areas are called a *trusted region* and an *untrusted region*, respectively. While the security guarantee of TEEs may vary depending on specific TEE designs, they generally provide the confidentiality and integrity of the trusted program[2], and such a security assurance is guaranteed by the hardware (i.e., the main processor). In order to provide the confidentiality, the main processor in a TEE ensures that an execution context of the trusted program (including registers and memory) is strictly isolated from other untrusted software components.

Remote attestation. Moreover, to provide integrity, TEE techniques support a security mechanism called remote attestation. The security of remote attestation is based on following two things: (i) the main processor is securely embedded with a cryptographic key during the manufacturing process for attestation keys; and (ii) every trusted software component can request a report called *software attestation* (called a *quote* in Intel SGX) including the hash value of itself to a trusted program responsible for measuring and signing (e.g., a Quote Enclave in Intel SGX).

An attestation key is provisioned by the hardware manufacturer in practice. For example, Intel provides the provisioning service (IPS) [46], and the Provisioning Enclave on the platform interacts with the IPS to get the EPID key, which is finally transferred to the Quote Enclave. Also, a device using the Samsung KNOX platform [77] has its private key called a device root key embedded at manufacturing, and the keys derived from the device root key are used for the attestation.

The main processor (in Intel SGX) or the trusted application (in ARM TrustZone) com-

[2] A program running in a TEE is called a trusted program since it is protected from the rest of the system.

putes an authenticated hash of the target program's memory footprint. That is, the hash value signed with the above attestation key is presented to a remote client who wishes to verify the integrity of the target program. The client that receives a quote can verify the integrity as follows [88]:

1. Validating the signature chain from the software attestation to the attestation CA (e.g., Intel CA),
2. Verifying that no keys have been revoked and the signing application is valid,
3. Confirming the status of the trusted computing base,
4. Comparing the enclave measurement in the quote with the expected value.

The 1st three steps are typically offloaded to an attestation verification service, but the last step should be done by the client (i.e., verifier). This means that if the verifier wants to perform remote attestation, she needs to request the verification of the received assertion to the attestation verification service and to know the expected value in advance. Note that this results in additional round-trips to the attestation verification service or require some method for a client to fetch an expected value of the edge application of interest.

Software attestation. Although the profile of the software attestation varies depending on the trusted computing platform, there are common fields including:

- **Measurement:** the hash value of the program to be measured. The value can be used to identify the program.
- **Device signature:** the signature generated by an edge device with an attestation key.
- **Nonce:** the random value to guarantee the freshness of the software attestation.

The above structure of the software attestation is somewhat simplified. For example, the software attestation in Intel SGX includes the hash of the Quote Enclave to guarantee the

authenticity of the signing enclave. The following is an example of the software attestation (SA) generated by a trusted computing platform P (i.e., SA_P) running an application A:

$$\mathsf{SA}_P = \mathsf{MR}_A \| \mathsf{Sign}\left(\mathsf{Priv}_P, \mathsf{Nonce} \| \mathsf{MR}_A\right),$$

where MR_A is a measurement value of an application A, Priv_P is a private key of a platform P, and $\mathsf{Sign}(sk, m)$ is a signature over m with a private key sk.

Among the various TEE systems, we focus on Intel SGX Data Center Attestation Primitives (DCAP) [88] and ARM TrustZone [6] to substantiate the SEED architecture.

Intel SGX DCAP. Intel SGX [5] is mainly designed for its hardware products, mostly processors for desktop/laptop computers and servers. That is, the TEE implementation in SGX focuses on enabling secure remote computation, which offloads a computational task from desktop/laptop clients to remote servers. Thus, Intel SGX originally uses the Enhanced Privacy Identifier (EPID) scheme to provide the privacy of the SGX machines by leveraging group key cryptography, which is far from our objective — visibility. On the other hand, Intel SGX DCAP is a general certification infrastructure for third party attestation and provides the ECDSA-based attestation, which authenticates the attester.

ARM TrustZone. On the contrary, ARM TrustZone is mainly designed for mobile and embedded computing platforms, where ARM-based processors dominate a market share. Hence, its design is to facilitate hardware-enforced isolation between the secure and non-secure worlds in a device. In ARM TrustZone, there is no reference model for the attestation. However, the Samsung KNOX platform [77] builds its own attestation infrastructure based on TrustZone and academic proposals [55, 44]. We also build the attestation system for ARM TrustZone detailed in §4.6.

4.2.3 TLS on the Third Party

Transport layer security (TLS) [23, 82] provides end-to-end security for Internet connections. One of the core functionalities of TLS is to allow a client to authenticate an ASP (e.g., domain

name) by checking whether the counterpart owns its private key, which corresponds to its public key on the certificate. Therefore, an edge application running over a large distributed infrastructure should authenticate itself as the ASP with its private key.

TLS with remote attestation. There are two representative proposals to combine TLS with remote attestation in the academic community – *Middlebox TLS (mbTLS)* and *SPX*.

In *Middlebox TLS (mbTLS)* [69], all the middleboxes have their own certificates. Each of the two endpoints authenticates their middleboxes individually and performs remote attestation on them individually during the mbTLS handshake. If the middleboxes are attested fine, the endpoints deliver the session keys to be used by the middleboxes.

SPX [12] is a Keyless SSL-style protocol that is enhanced with remote attestation. When a client wants to establish a TLS session with a server, an edge device in the middle sends its attestation proof to a server. The server attests the edge device, and if successful, sends the shared session key to the edge device, which can then participate in the session.

Note that both protocols require a server to be involved in the session establishment. Furthermore, the process of the attestation verification is unclear in the sense that they do not specify how the attestation is verified with or without a remote attestation service.

Discussions on protocols for edge computing platform. From the above introductions, our approach to combining the TLS protocol with remote attestation without sharing private keys is designed with the following rationale.

First, the current proposals for inserting a private key into a Hardware Security Module (HSM) [2, 33, 1] have limitations. Even if we can completely trust the HSM, we believe removing private keys of ASPs from edge devices (as in Keyless SSL) is much safer. This is because an edge device is physically accessible; thus, there can be many attempts to perform side-channel attacks targeting the security modules (e.g., [102]). Furthermore, this requires a method to bind the attestation key and the ASP's private key; Without such methods, there can be the TOCTTOU attack and the Cuckoo attack as described in [12].

Second, there is a trade-off between security/deployability and performance, the latter

of which is prioritized in practice. Keyless SSL, mbTLS, and SPX require each edge server (or middlebox) to manage its private key. However, the performance of these schemes is degraded compared with the current practice since an ASP should be involved in the session establishment. On the other hand, DC performs better but requires the edge servers to manage short-term private keys for individual domain clients to be upgraded to support the protocol. To address the security and deployability issues of DC, Keyless SSL can be deployed as a fallback to DC. That is, if a client does not support the DC protocol, Keyless SSL will be used instead. Thus, we consider not only how to achieve better performance but also how to provide a fallback mechanism and how to reduce the number of secrets.

Finally, the process of attestation verification should be clarified. We find that both mbTLS and SPX do not detail how to verify attestations. We argue that the following two points should be made clear in the remote attestation.

- The verification of the attestations requires round-trips to the attestation verification service. Note that the API is provided with HTTPS [81] in the case of the Intel Attestation Service (IAS). This means that if a client uses the IAS as a verifier during its TLS handshake, it requires another TLS session establishment with the IAS, which increases the latency. Therefore, the problem of the attestation verification service should be addressed.

- Obtaining the expected value of the enclave measurement in advance by a client is challenging. As described in §4.2.1, the client connects to an edge application without being aware that she is communicating with an edge device. Furthermore, the number of edge applications may be unlimited in the edge computing platform. Thus, it might be impossible for the client to manage all the measurement values of the edge applications.

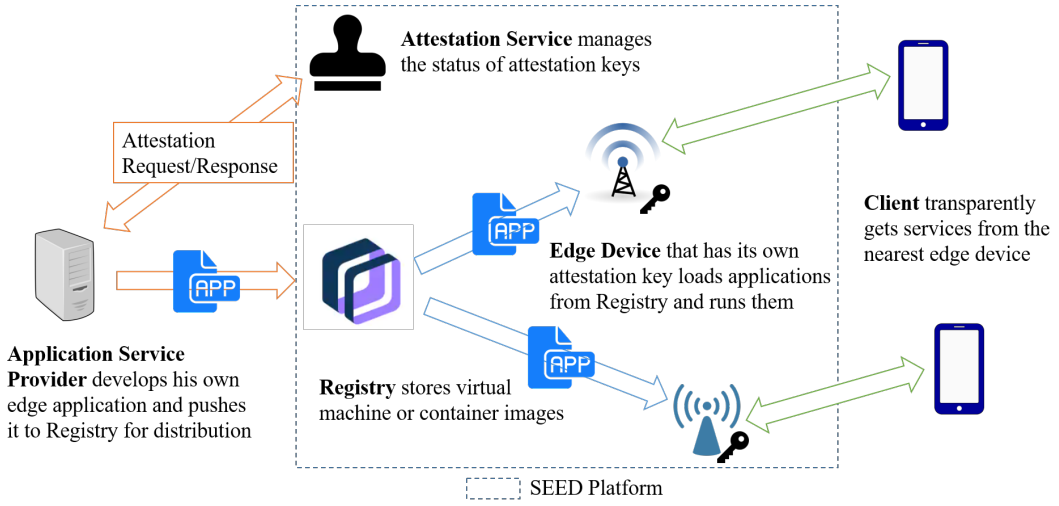


Figure 4.1: SEED Platform Scenario. In the SEED platform that consists of SEED devices, an attestation service, and a registry, an application service provider (ASP) develops his edge application and registers it to the registry. The application is distributed to edge devices that have their own attestation keys on the platform. A client that wants to connect to an ASP transparently is redirected to the nearest edge device and gets service from the edge device.

4.3 SEED Overview

In this section, we describe a high-level overview of the SEED platform, which is shown in Figure 4.1.

Participants. A SEED *platform* may have multiple SEED *devices* on which *edge applications* are loaded and running. A SEED device can generate a software attestation leveraging its hardware security module. We assume that every SEED device has its own public/private key pair, which can be provisioned as described in §4.2.

The SEED platform also has two components: a registry and an attestation service. A *registry* manages edge applications in the form of virtual machine or container images. When a request for a particular application is received, the registry provides the image to the requesting SEED device. An *attestation service* manages the status of attestation keys (i.e., public keys of SEED devices) like the e.g., revocation status of the keys.

With the SEED platform, an *application service provider (ASP)* provides services at the user's proximity. He develops edge applications and registers them with the registry in the platform; he need not consider how to distribute the edge applications. At the time when an application service provider registers his application, he receives a certificate of the attestation service run by the platform and configures the attestation service as the trusted one by inserting the certificate into his trusted certificate store. We call a user or her program (e.g., a browser) that uses edge services a *client*.

Adversary capabilities. Participants in our system are exposed to an adversary who controls the untrusted part of the system. The adversary may also have full control of a network, which means that he can capture all the messages on the network. He can also modify, inject, and drop packets in the network. We also assume an adversary can mount an attack against the software stack in the untrusted region and can perform Iago attacks [17] by giving an invalid result of syscalls to the trusted application. However, the adversary is computationally bounded, which means that he cannot break cryptographic primitives. The booting process for the trusted environment is secure. Denial-of-service attacks and side-channel attacks on TEEs are out-of-scope.

Solution overview. To resolve the issues as discussed in §4.2, we require an ASP to provide its clients with sufficient information about an edge platform. In this way, the ASP allows a client to verify the integrity of its edge application, while not embedding his private key into the SEED platform. We believe this is the ASP's responsibility since many privacy regulations, such as the General Data Protection Regulation (GDPR), require ASPs to take *accountability* with appropriate technical measures [27].

To this end, we propose that an ASP issue a CROSS CREDENTIAL (CC) to a trusted SEED device, which can attest the loaded edge application of the ASP. The CC (i) allows the ASP not to share his long-term private keys, and (ii) provides the following information to a client.

- A validity time that is granted by an ASP

- A attestation key of a device that an ASP authorizes
- A policy of an ASP (e.g., a URL that describes an edge application)
- An expected value (e.g., hash) that identifies an edge application developed by an ASP.

4.4 SEED Design

In this section, we present how SEED is designed. We begin with our security goals. Then we detail a CC, a short-term data structure to achieve the security goals. Next, we discuss the TLS-SEED protocol to allow a client, an edge device, and an ASP to communicate with each other while satisfying the goals. Let us then explain the advantages of a CC.

4.4.1 Security Goals

SEED aims to provide (i) *accountability* for ASPs and (ii) *visibility* for clients, while achieving (iii) minimal *performance* overhead compared with the current CDN practice.

- **Accountability:** In the current distributed system such as CDNs, an *application service provider (ASP)* has no other choice but to trust the management of the system without any technical assurance. For trustworthy services, the ASP may want to attest his application deployed over the third party platform (G1 Authorization after attestation). Note that this authorization should be performed without sharing his private key (G2 No private key sharing) while the number of secrets managed by a SEED platform should be kept minimal (G3 Minimal secrets).
- **Visibility:** A *client* is not aware of a middle node (i.e., edge device) that can store her sensitive data in the current practice. We propose that she should know (i) if such a middle node exists, and (ii) whether it is authorized by an ASP (G4 Trusted device authentication). Furthermore, she should be able to attest an edge application to confirm whether the application behaves as authorized by the ASP (G5 Edge application attestation).

- **Performance:** Recall that many prior studies make cloud servers involved during handshaking, which significantly increases the performance overhead compared with the current CDN practice. The application service delay experienced by a client should be minimal (G6 Minimal performance overhead).

4.4.2 Cross Credential (CC)

To achieve the above security goals, we introduce a short-term proof, called a CROSS CREDENTIAL (CC). The CC proves that an edge application is trustworthy and running on a SEED device is authorized by an ASP. The structure of a CC can be expressed as follows; its notation is provided in Table 4.1:

$$\mathsf{CC}^{S}_{Dev} = \mathsf{m}_2 \| \mathsf{Sign}_S \left(\mathsf{Priv}_S, \mathsf{m}_2 \right)$$

where,

$$\mathsf{binding} = \mathsf{m}_1 \| \mathsf{Sign}_{Dev} \left(\mathsf{Priv}_{Dev}, \mathsf{m}_1 \right)$$

$$\mathsf{m}_1 = H \left(\mathsf{Pub}_S \| \mathsf{MR}_{App} \right)$$

$$\mathsf{m}_2 = \mathsf{not\_before} \| \mathsf{not\_after} \| \mathsf{binding}$$

A binding consists of a message m_1 and its signature with the private key of a SEED device, where m_1 is the hash value of two elements: (i) the public key of an ASP (Pub_S) and (ii) the measurement of an edge application (MR_{App}). This means that the SEED device with its private key (Priv_{Dev}) loads an edge application, which is identified with the measurement value (MR_{App}) of the application. The edge application is developed by the ASP (i.e., the owner of the public key Pub_S). The not_before and not_after, which indicate the validity period of a CC, are prepended to binding, resulting in m_2. Finally, m_2 is signed by the ASP S, which is CC^{S}_{Dev}. Note that the CC is used within the validity time and until the public key of the SEED device or that of the ASP is changed.

Notation	Meaning
ASP	An application service provider (ASP)
C	A client
Dev	An edge device
$rand_e$	A random value generated by an entity e
MR_e	The measurement value of an entity e
SA_e	The software attestation signed by an entity e
CC_f^e	The cross credential between an entity e and an entity f, meaning that an entity e authorizes its service over an entity f
Pub_e	An entity e's public key
$Priv_e$	An entity e's private key
m_i	The message indexed with i
$H(m)$	The hash of message m
$Sign(k, m)$	The signature of message m with key k
$Sign_e$	The signature generated by an entity e
$a\|\|b$	a concatenated with b

Table 4.1: Notation used in the description of TLS-FRONTEND-SEED and TLS-BACKEND-SEED.

4.4.3 TLS-SEED: TLS extensions for SEED

TLS-SEED is a set of protocols to establish a secure session with a SEED device. Specifically, TLS-SEED consists of two protocols: (i) TLS-BACKEND-SEED between the SEED device and a cloud server operated by an ASP (i.e., the backend channel) and (ii) TLS-FRONTEND-SEED between a client and the SEED device (i.e., the frontend channel). We detail each protocol in the following. Note that both protocols are based on both TLS 1.2 [23] and TLS 1.3 [82].[3]

TLS-BACKEND-SEED

The main objectives of TLS-BACKEND-SEED are (i) to allow ASPs to take accountability — authorization after attestation (G1), (ii) not to share the private keys of the ASPs with edge devices (G2), and (iii) to keep minimal secrets in the edge computing platform (G3) — by issuing CCs. The protocol is executed whenever a SEED device loads the application or the validity time in the CC is expired. The overall process is shown in Figure 4.2.

[3]The way to extend the TLS handshake is defined in [23] with the extension message in `ClientHello` and `ServerHello`.

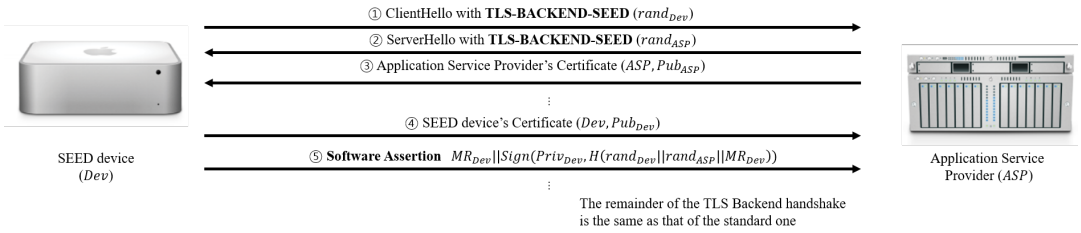


Figure 4.2: **TLS-BACKEND-SEED.** An ASP performs remote attestation during the handshake and authorizes the SEED device by updating another CC after the handshake (G1). Thanks to the CC, the ASP need not transfer his private key to the SEED device (G2). Note that the CC is public; thus, the SEED device only needs to protect its own private key (G3).

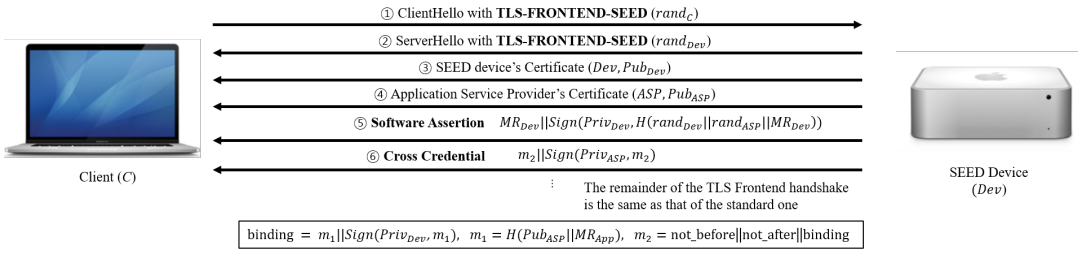


Figure 4.3: **TLS-FRONTEND-SEED.** This process includes the presentation of a CC. The CC helps the client verify (i) the communication counterpart is the intended edge application on the edge device authorized from the ASP (G4), and (ii) the edge application will behave as expected (G5). Note that this protocol does not add any additional round-trip messages as similar to the current CDN practice (G6).

1. A SEED device initiates this protocol with `ClientHello` including a special constant (say, `TLS-BACKEND-SEED`) to indicate its support of the protocol (①).
2. An ASP responds with `TLS-BACKEND-SEED` in `ServerHello` to agree on the usage of the protocol (②).
3. Then, the ASP sends (i) `CertificateRequest` to request the SEED device's certificate and (ii) `Certificate` to deliver its certificate (③).
4. The SEED device validates the ASP's certificate and checks whether the ASP is one of the contractors.

5. If successful, the SEED device sends `Certificate` that includes its certificate (④), along with the software attestation (⑤). For the nonce value, the hash value of the handshake messages until this point is used to bind the software attestation to this session and to guarantee the freshness.
6. The ASP validates the SEED device's certificate and checks whether the SEED device can be trusted. Then, the ASP verifies the signature in the software attestation with the help of the attestation service in the edge computing platform.
7. If successful, the ASP compares the measurement value in the software attestation with the expected value. Note that the ASP knows the expected value since he deploys the edge application.
8. Finally, both proceed with the remainder of the protocol and establish the session key.

After establishing the backend session, the SEED device and the ASP cooperate to generate a CC according to the following procedure:

1. The SEED device generates a hash value by concatenating the ASP's public key and the measurement of the deployed edge application.
2. The SEED device signs the hash with its private key and sends the signature to the ASP. We call this value CCRequest.
3. Then the ASP verifies the signature and checks the binding value in CCRequest.
4. If the verification succeeds, the ASP prepends the validity time to CCRequest and finally signs it, resulting in a CC.
5. Finally, the ASP sends the CC back to the SEED device.

TLS-FRONTEND-SEED

The purpose of TLS-FRONTEND-SEED is to provide visibility for clients – trusted device authentication (G4) and edge application attestation (G5) – without adding substantial

overheads (G6). Note that TLS-FRONTEND-SEED provides a method for a client to attest the edge applications even if the client does not know the hash values of the edge applications.

Figure 4.3 illustrates the process of TLS-FRONTEND-SEED as follows:

1. The client begins with this protocol by sending `ClientHello` including `TLS-FRONTEND-SEED` to use TLS-FRONTEND-SEED if possible (①).
2. On intercepting the message, the SEED device examines the `ClientHello` extensions to check TLS-FRONTEND-SEED. If so, the SEED device includes `TLS-FRONTEND-SEED` in `ServerHello` to indicate its support (②).
3. Then, the SEED device also provides its certificate (③) and the ASP's certificate (④).
4. Next, the SEED device sends the software attestation (⑤) of which the nonce is the hash value of the handshake messages, followed by the CC (⑥).
5. On receiving the two certificates along with the software attestation and the CC, the client verifies them as follows:
   - (Certificate validation) The client initially validates both certificates and checks the name in the ASP's certificate.
   - (software attestation validation) The client then verifies the signature in the software attestation. If the signature is proven to be valid, she can confirm that (a) the software attestation is originated from the SEED, (b) it is freshly generated, and (c) the measurement value is not altered. Note that at this point, the client cannot know whether the measurement value is correct or not. She will get the measurement value from the CC later on.
   - (Signature verification in CC) Next, the client verifies two signatures in the CC. By confirming the SEED device's signature and the ASP's signature, the client can figure out the communicating counterpart is the edge application running on the SEED device authorized by the ASP.
   - (Validity time checking) After the verification, the client checks the validity time. If the

current time is not within the validity time, she aborts the session; otherwise, we proceed to the next step.

- (binding verification) The client finally extracts the public key from the ASP's certificate, concatenates it with the measurement value from the software attestation, and applies the hash function to the concatenated data. If the hash value matches the binding in the CC, the client confirms that (a) the ASP deploys the program that is identified by its measurement value and (b) the measurement value reported by the SEED device is correct.

6. The remainder of the TLS-SEED handshake proceeds as the standard TLS handshake, and the frontend session is finally established.

Other issues

For session resumption, we streamline TLS-FRONTEND-SEED by removing the certificates. It is sufficient for the client only to verify the software attestation since the client has already received a pre-shared key (from the SEED device), which is used for the session resumption.

We assume a fallback mechanism for a client that does not support TLS-FRONTEND-SEED. In such cases, the client sets up a connection with the ASP while the SEED device serves as a relay.

We want to highlight two points regarding the fallback mechanism. First, the session established by the fallback mechanism should not allow the SEED device to inspect the messages between the ASP and the client. As the SEED device is not involved in the handshake and cannot know the session key between the ASP and the client, the SEED device cannot read the messages.

Second, the fallback mechanism should not add substantial overhead. Since the SEED device serves as an application-layer relay, this might add some computation overhead on the device. Our testbed experiments show the overhead is marginal in terms of the latency and the CPU computation to be detailed in §4.6.

4.4.4 Implications of Cross Credential

Contributions of CC. First, the SEED platform should provide authorization after attestation (G1). A CC can be a time-bounded authorization token by an ASP. Recall that the ASP attests his edge application during the handshake. After this, he can deauthorize the service due to the compromised platform or other reasons based on his policy.

Second, the ASP should not share his private key with the SEED platform (G2). A SEED device does not impersonate the ASP. Instead, the SEED device proves the integrity of the application with the CC.

Third, the number of secrets should be minimal on a SEED platform (G3). Using CCs does not increase the number of secrets kept in a SEED device, which protects its private key only.

Fourth, the client can figure out the counterpart is a trusted computing platform (i.e., a trusted device) authorized by the ASP (G4). Note that only a device that participates into the procedure of a CC generation can use the CC since CC includes a device's unique key. Therefore, other devices cannot use the CC to impersonate a particular device. The CC indicates that the ASP deploys its application on a trusted device in the SEED platform, and authorizes the device to run the application.

Lastly, the client should be able to attest the remote edge application to ensure that it is running as expected without compromises (G5). Since the CC includes the hash value of the edge application, the client obtains the expected value for the attestation and compares it with the measurement value in the software attestation. In this way, the client can perform remote attestation for the third party.

Discussions of CC. An ASP can always control the permission to let the SEED platform provide the service on his behalf. This is mainly controlled by two components in a CC: (i) the validity period and (ii) certificate revocation.

(1) When signing the binding received from the SEED platform, the ASP can decide

the validity period of the CC. The longer is the validity period, the less frequently is a CC updated. However, he may not be able to counter security incidents earlier. (2) If the ASP is compromised, malicious services could be deployed at the SEED platform. Once the administrator of the ASP notices the incident, his certificate can be immediately revoked. Since the CC is also signed by his certificate, revoking either his certificate or the device' certificate may immediately stop the effect of the compromised edge application[4]. Note that due to this feature, the revocation mechanism of CCs is not needed.

4.5 Security Analysis

In this section, we analyze the security of the TLS-FRONTEND-SEED by which a client and an edge device set up a TLS connection. To this end, we extend the authenticated confidential channel establishment (ACCE) model to support TLS-FRONTEND-SEED, resulting in ACCE-SEED. Based on the extended model, we formally prove that the TLS-FRONTEND-SEED is ACCE-SEED-secure.

4.5.1 Overview of ACCE

The ACCE model originally used in the security proof of the TLS 1.2 protocol does not guarantee indistinguishability in the authenticated key exchange model [43]. Instead, it focuses on the channel established with a session key. The model separates the pre-accept phase (i.e., the handshake protocol) for entity authentication and the post-accept phase (i.e., the record protocol) for channel security. The model is then used to prove that the session key established after entity authentication in the pre-accept phase constructs a secure channel used in the post-accept phase. Since CCs can also be used in TLS 1.2, the method of entity authentication in TLS needs to be revised. Thus, we extend the ACCE model to the ACCE-SEED model to prove the security of TLS-SEED.

[4]The client is responsible for checking the the revocation status of both certificates.

4.5.2 ACCE-SEED Protocol Execution Environment

Participants, sessions, and attributes. The environment consists of n_P participants $P_1, \cdots, P_{n_P}$, each of which is one of $\mathcal{C}$ (Clients), $\mathcal{DEV}$ (SEED devices), or $\mathcal{ASP}$ (ASPs). We use $\mathcal{P}$ for a set of all the participants (i.e., $\mathcal{P} = \mathcal{C} \cup \mathcal{DEV} \cup \mathcal{ASP}$). Also, all the participants in $\mathcal{DEV}$ and $\mathcal{ASP}$ have their long-term private/public key pairs. Moreover, each participant has the following variables, some of which can be set to none, $\perp$. Note that for an n-tuple $p = (c_1, \cdots, c_n)$, we use $p.c_i (i \in \{1, \cdots, n\})$ to denote an i-th component in the n-tuple.

- sk: a private key of a participant. A client may not have her own key pair; thus, it is set to $\perp$.
- pk: a public key of a participant. A client may not have her own key pair; thus, it is set to $\perp$.
- CertSet: an indexed set of certificates.
- MRSet: a set of measurement values, each of which is denoted by mr_j^i, a hash of an edge application deployed by ASP P_i over device P_j.
- CCSet : a set of "valid" cross credentials, each of which is denoted as cc_j^i, which is a CC issued by ASP P_i to device P_j. Note that a CC is expressed as a 5-tuple, $cc_j^i = (t_s, t_e, P_i.pk, mr_j^i, \sigma_j, \sigma_i)$. t_s and t_e are the starting ending times of the CC, respectively. where $P_i \in \mathcal{ASP}$, $P_j \in \mathcal{DEV}$. Also, $\sigma_j = \mathsf{Sign}(P_j.sk, m_1)$, where $m_1 = \mathsf{H}(P_i.pk \| mr_j^i)$, and $\sigma_i = \mathsf{Sign}(P_i.sk, m_2)$, where $m_2 = t_s \| t_e \| m_1 \| \sigma_j$.

Each participant P_i can run concurrent sessions. We denote the s-th session of P_i by π_i^s and the maximum number of the sessions per participant is n_S. The session π_i^s can access the long-term key of P_i and maintains the state of the session with the following variables:

- $\rho \in \{\mathsf{init}, \mathsf{resp}\}$: The role of the party in the session. The session of init initiates the

handshake. On the other hand, the session of resp responds to the initial message from the session of init.

- $\alpha \in \{\mathsf{inprogress}, \mathsf{reject}, \mathsf{accept}\}$: The status of the session.
- k: The session key established after the key exchange.
- pid: The communication counterpart.
- tid: The intended ASP. For sessions of participants in $\mathcal{DEV}$ or $\mathcal{ASP}$, this is set to the intended target of the client.
- sid: The session identifier. In the TLS protocol the session identifier is a transcript of the handshake messages like a tls_unique value in [108].
- $timestamp$: The timestamp when the session is established.
- sa: The software attestation used during the handshake. It is denoted by $sa^i_j = (\mathsf{nonce}, mr^i_j, \sigma_j)$, where nonce is the concatenation of two random values, the hash of the handshake messages until this point, mr^i_j is a measurement value of an edge application deployed to P_j by P_i, and σ_j is a signature generated by P_j over nonce and mr^i_j.
- cc: The cross credential used in the session.
- st_E, st_D: States for the stateful authenticated encryption and decryption algorithms.
- b: A hidden bit (used for a security experiment). This value is initialized with $b \xleftarrow{\$} \{0, 1\}$.

Capabilities of an adversary. As described in §4.3, we assume that a Dolev-Yao adversary can fully control the network, meaning that it can read, reorder, create, alter, and drop messages. During the handshake, the adversary can interact with sessions by using the Send queries as described below:

- $\mathsf{Send}(\pi_i^s, m) \rightarrow m'$: The adversary sends this query with message m to π_i^s. P_i processes message m, updates the state of π_i^s, and generates message m' if needed, following the protocol. The query returns an error symbol $\perp$ if the session is already established (i.e., $\pi_i^s.\alpha = \mathsf{accept}$).

The adversary can also leak the private key or the session key with the two queries.

- $\mathsf{Reveal}(\pi_i^s) \rightarrow k$: With this query, the adversary can get the session key (i.e., $\pi_i^s.k$).
- $\mathsf{Corrupt}(P_i) \rightarrow sk$: If $P_i.sk = \perp$, it returns $\perp$. Otherwise, with this query, the adversary can get the long-term private key of P_i.

After the secure channel is established, the adversary can encrypt/decrypt a message with the following two oracles that perform encryption/decryption with the shared key established during the handshake [15].

- $\mathsf{Encrypt}(\pi_i^s, m_0, m_1) \rightarrow C$: If $\pi_i^s.k = \perp$, the query returns $\perp$. Otherwise, it performs authenticated encryption with additional data (AEAD) encryption on both m_0 and m_1 (the resultant ciphertexts are C_0 and C_1 respectively), updates states of a session, and returns one of the ciphertexts according to the result of tossing a coin. The algorithm is described in Algorithm 1.
- $\mathsf{Decrypt}(\pi_\mathsf{i}^\mathsf{s}, \mathsf{C}) \rightarrow m$ or $\perp$: If $\pi_i^s.k = \perp$, the query returns $\perp$. Otherwise, it performs AEAD decryption on C, checks states of the session, and returns the decrypted message if there is no error. The algorithm is described in Algorithm 2.

Note that the above encryption and decryption oracles capture *indistinguishability under chosen ciphertext attack*, *integrity of ciphertexts*, *integrity of associated data*, and *stateful delivery of ciphertexts*.

Finally, we add two oracles called RegParty and Deploy. RegParty is a modified version of the new party-registration oracle in [13] and Deploy is an oracle to capture the behavior of deploying an edge application into an edge device by an ASP.

- RegParty(P_i, role): this oracle registers P_i as a role $\in$ {"ASP", "Dev"}. Then, a private key, a public key, and a certificate are installed to P_i, while the adversary gets a certificate and a public key.

- Deploy(P_i, mr_j^i, P_j): with this query where $P_i \in \mathcal{ASP}$ and $P_j \in \mathcal{DEV}$, mr_j^i is added to P_i.MRSet and P_j.MRSet. Furthermore, cc_j^i is added to P_i.CCSet and P_j.CCSet.

Matching sessions. We call π_j^t matches π_i^s if two conditions are met: (i) $\pi_i^s.\rho \neq \pi_j^t.\rho$ and (ii) $\pi_i^s.sid$ is a prefix of $\pi_j^t.sid$ or equal to $\pi_j^t.sid$.

Session freshness in ACCE-SEED. We call π_i^s of P_i is *fresh* with a counterpart P_j when the following conditions hold:

- On the last query of the adversary, π_i^s has finished its handshake by $\pi_i^s.\alpha = \mathsf{accept}$.

- No Corrupt query was made to P_i, $\pi_i^s.pid$, and $\pi_i^s.tid$.

- No Reveal query was made to π_i^s and π_j^t where $\pi_i^s.sid = \pi_j^t.sid$

Correctness of ACCE-SEED. For two communicating oracles π_i^s of P_i ($P_i \in \mathcal{C}$) and π_j^t of P_j ($P_j \in \mathcal{DEV}$), we say the ACCE-SEED protocol is *correct* if, in the presence of a benign adversary, the following conditions hold:

- $\pi_i^s.\alpha = \mathsf{accept}$ and $\pi_j^t.\alpha = \mathsf{accept}$

- $\pi_i^s.k = \pi_j^t.k \in \mathcal{K}$,

where $\pi_i^s.k$ and $\pi_j^t.k$ are session keys generated by π_i^s and π_j^t, respectively.

4.5.3 ACCE-SEED Security

We extend the ACCE model and define three security properties for ACCE-SEED-security: trusted device authentication, edge application attestation, and (front-end) channel security.

Security experiment. In the ACCE-SEED security experiment, there is an adversary $\mathcal{A}$ against a challenger. Then, the adversary uses the queries defined in §4.5.2 to break the TLS-FRONTEND-SEED protocol Π and to win the following games.

DEFINITION 1 **(Trusted device authentication (TDA))** When TLS-FRONTEND-SEED is executed, a client should be able to authenticate the SEED device equipped with a hardware security module and authorized by the intended ASP. In this game, the adversary $\mathcal{A}$ can query the oracle RegParty, Deploy, and other ACCE oracles. We say the adversary wins the TDA game if there exists a session π_i^s $(P_i \in \mathcal{C})$ in accept state without a matching session π_j^t from partner $P_j \in \mathcal{DEV}$ that provides services on behalf of $P_k \in \mathcal{ASP}$. This can be formalized as follows:

- $\pi_i^s.\rho = \mathsf{init}$.
- $\pi_i^s.\alpha = \mathsf{accept}$.
- $\pi_i^s.tid = P_k$.
- Neither $\mathsf{Corrupt}(P_j)$ nor $\mathsf{Corrupt}(P_k)$ has been issued before π_i^s enters accept state.
- $\mathsf{RegParty}(P_k, \text{“}ASP\text{”})$, $\mathsf{RegParty}(P_j, \text{“}Dev\text{”})$, and

  $\mathsf{Deploy}(P_k, mr_j^k, P_j)$ have been queried, but there is no matching session π_j^t such that $\pi_i^s.sid = \pi_j^t.sid$.

$\mathtt{Adv}_{\Pi}^{\mathsf{tda}}(\mathcal{A})$ is defined as the adversary's winning probability, where the probability is taken over the random coins of all the participants.

DEFINITION 2 **(Edge application attestation (EAA))** When TLS-FRONTEND-SEED is executed, a client wants to confirm whether the counterpart is an edge application deployed by the ASP. We say the adversary wins the EAA game if there exists a session π_i^s $(P_i \in \mathcal{C})$ in accept state and the following conditions simultaneously hold for $P_j \in \mathcal{DEV}$ and $P_k \in \mathcal{ASP}$:

- $\pi_i^s.\rho = \mathsf{init}$
- $\pi_i^s.\alpha = \mathsf{accept}.$
- $\pi_i^s.pid = P_j$
- $\pi_i^s.tid = P_k$
- $\pi_i^s.sa.mr = mr_j^k$
- $\pi_i^s.cc.mr = mr_j^k$
- No Corrupt query is made to P_j and P_k
- Deploy(P_k, mr_j^k, P_j) query has been issued, but there is no matching session π_j^t such that $\pi_i^s.sid = \pi_j^t.sid$.

DEFINITION 3 **(Channel security (CS))** In a channel security game, the adversary $\mathcal{A}$ uses oracles arbitrarily and outputs a tuple (π_i^s, b). He breaks the channel security if the following conditions simultaneously hold:

- π_i^s is fresh with the counterpart $P_j \in \mathcal{DEV}$.
- $\pi_i^s.\mathsf{b} = b'$.

$\mathtt{Adv}_{\Pi}^{\mathsf{cs}}(\mathcal{A})$ is defined as the adversary's winning probability, denoted by $|p - \frac{1}{2}|$, where p is the probability that $\mathcal{A}$ correctly answers the encryption challenge (i.e., , $b = b'$).

4.5.4 Security Result

We state the following theorem to prove the security of TLS-FRONTEND-SEED.

THEOREM 1 Let Π be TLS-FRONTEND-SEED. If a signature scheme used in Π satisfies existentially unforgeable under chosen message attacks (Definition 4) and a secure hash function used in Π satisfies collision-resistant (Definition 5), then Π is ACCE-SEED-secure.

The following lemmas are stated to prove Theorem 1.

LEMMA 1 Let Π be TLS-FRONTEND-SEED. If a signature scheme used in Π satisfies existentially unforgeable under chosen message attack (Definition 4) and a secure hash function used in Π satisfies collision-resistant (Definition 5), then Π provides *trusted device authentication* as well as *edge application attestation* with the following adversary's advantage:

$$\begin{aligned}\mathtt{Adv}_{\Pi}^{\mathsf{tda,eaa}} &\leq \frac{(n_P n_S)^2}{2^{\ell}} \\ &+ n_P n_S \cdot (n_P \cdot \epsilon_{\mathsf{euf\text{-}cma}} + n_P n_S \cdot (n_P \cdot (\epsilon_{\mathsf{euf\text{-}cma}} + 2\epsilon_{\mathsf{coll}})))\end{aligned}$$

Proof. The proof is based on a sequence of games [94]. Let break_{δ} be the event that the adversary wins the game in GAME δ, and abort_{δ} be the event that the challenger aborts in GAME δ.

GAME 0. **[Original experiment]** This game is equal to the ACCE-SEED experiment described in §4.5.3; thus,

$$\mathtt{Adv}_{\Pi}^{\mathsf{tda,eaa}}(\mathcal{A}) = \Pr[\mathsf{break}_0]$$

GAME 1. **[Exclude colliding nonces]** We add an abort rule. A challenger aborts if a nonce is repeatedly used. Thus, the following is satisfied with ℓ-bit nonces.

$$\Pr[\mathsf{break}_0] \leq \Pr[\mathsf{break}_1] + \tfrac{(n_P \cdot n_S)^2}{2^{\ell}}$$

GAME 2. **[Identify accepted client]** We add an abort rule. A challenger guesses a pair $(i^*, s^*) \xleftarrow{\$} (n_P, n_S)$ and aborts the game if $\pi_{i^*}^{s^*}$ has $\pi_{i^*}^{s^*}.\rho \neq \mathsf{init}$ or $\pi_{i^*}^{s^*}$ is not the first session that satisfies DEFINITION 1 or DEFINITION 2. The probability that a challenger guesses the correct pair is $\frac{1}{n_P n_S}$; thus, the following is satisfied:

$$\Pr[\mathsf{break}_1] = n_P n_S \cdot \Pr[\mathsf{break}_2]$$

GAME 3. **[Signature forgery (SEED device)]** We add an abort rule. A challenger aborts the game if a session $\pi_{i^*}^{s^*}$ receives a SEED's certificate indicating j and a signature for a session key material is valid, but there is no session that sends the signature.

$$\Pr[\mathsf{break}_2] \leq \Pr[\mathsf{break}_3] + \Pr[\mathsf{abort}_3]$$

Note that the event abort_3 can only occur if a signature is forged by an adversary. For each entity, the success probability to forge the signature by an adversary is $\epsilon_{\mathsf{euf\text{-}cma}}$; thus, $\Pr[\mathsf{abort}_3] \leq n_P \cdot \epsilon_{\mathsf{euf\text{-}cma}}$. Therefore, we get:

$$\Pr[\mathsf{break}_2] \leq \Pr[\mathsf{break}_3] + n_P \cdot \epsilon_{\mathsf{euf\text{-}cma}}$$

GAME 4. **[Identify communicating peer]** We add an abort rule. In this game, a challenger guesses a pair $(j^*, t^*) \xleftarrow{\$} [n_P] \times [n_S]$ that represents a communicating session. Then, a challenger aborts the game if the following Send-queries have never been issued.

- $\mathsf{Send}(\pi_{i*}^{s^*}, \perp)$ that returns a message m_0 with a client random,
- $\mathsf{Send}(\pi_{j*}^{t^*}, m_0)$ that returns a message m_1 with a signature including a client random.

The probability that a challenger guesses the right answer is $\frac{1}{n_P n_S}$; thus, we have

$$\Pr[\mathsf{break}_3] = n_P n_S \cdot \Pr[\mathsf{break}_4]$$

GAME 5. **[Signature forgery (ASP)]** In this game, the game proceeds as before, but we add an abort rule. A challenger aborts the game if a session $\pi_{i*}^{s^*}$ receives an application service provider's certificate indicating k with CC_{j*}^{k}, $\pi_{i*}^{s^*}.timestamp \in [t_s, t_e]$, both $\mathrm{CC}.\sigma_{j*}$ and $\mathrm{CC}.\sigma_k$ are valid, but P_k has never generated $\mathrm{CC}_{j*}^{k}.\sigma_k$. Technically, the event abort_5 happens when an adversary forges the signature with the success probability $\epsilon_{\mathsf{euf\text{-}cma}}$ for each entity. Thus:

$$\begin{aligned}\Pr[\mathsf{break}_4] &\leq \Pr[\mathsf{break}_5] + \Pr[\mathsf{abort}_5] \\ &\leq \Pr[\mathsf{break}_5] + n_P \cdot \epsilon_{\mathsf{euf\text{-}cma}}\end{aligned}$$

GAME 6. **[Identify an ASP]** In this game, a challenger proceeds the game as before, but we add an abort rule. A challenger guesses $k^* \xleftarrow{\$} [n_P]$ and aborts the game if $\pi_i^s.tid = P_{k^*}$, but $\mathsf{Deploy}(P_{k^*}, mr_{j^*}^{k^*} P_{j^*})$ has never been issued (i.e., $mr_{j^*}^{k^*} \notin P_{k^*}.\mathsf{MRSet}$), thus:

$$\Pr[\mathsf{break}_5] = n_P \cdot \Pr[\mathsf{break}_6]$$

GAME 7. **[Remove colliding edge applications]** We add an abort rule. A challenger aborts the game if the measurement value reported by a SEED and the expected value in the binding $(\mathrm{CC}_{j^*}^{k^*}.b_{j^*}^{k^*})$ are the same, but the measured application and the deployed application are different. The abort_7 is an event that a hash collision happened; thus, $\Pr[\mathsf{abort}_7] \leq \epsilon_{\mathsf{coll}}$. Therefore, we get:

$$\Pr[\mathsf{break}_6] \leq \Pr[\mathsf{break}_7] + \epsilon_{\mathsf{coll}}$$

GAME 8. **[Remove the hash collision]** We add an abort rule. A challenger aborts the game if the hash collision happened such that $\pi_{i^*}^{s^*}.sid = \pi_{j^*}^{t^*}.sid$ but the messages that make $\pi_{i^*}^{s^*}.sid$ and $\pi_{j^*}^{t^*}.sid$ are different.

$$\Pr[\mathsf{break}_7] \leq \Pr[\mathsf{break}_8] + \epsilon_{\mathsf{coll}}$$

Note that in GAME 8, the conditions in DEFINITION 1 and DEFINITION 2 cannot be satisfied; thus, $\Pr[\mathsf{break}_8] = 0$

Taking all the above probabilities in the games together, LEMMA 1 is proved. □

LEMMA 2 Let Π be TLS-FRONTEND-SEED and $\mathcal{A}$ be an adversary in the channel security game. If a pseudorandom function PRF is prf-secure, a stateful length hiding authenticated encryption is slhae-ecure, and the DDH-problem is hard with respect to G, then Π provides the channel security.

$$\mathtt{Adv}^{\mathsf{enc}}_{\Pi} = \epsilon_{\mathsf{enc}} \leq \epsilon_{\mathsf{tda,eaa}} + n_P n_S \cdot (\epsilon_{\mathsf{ddh}} + 2 \cdot \epsilon_{\mathsf{prf}} + \epsilon_{\mathsf{slhae}})$$

Proof. Let guess_δ be the event that the adversary guesses the correct answer, i.e., $\pi^s_i.\mathsf{b} = b$ in GAME δ, and abort_δ be the event that the challenger aborts in GAME δ.

GAME 0. **[Original experiment]** This game equals to ACCE-SEED experiment described in §4.5.3; thus,

$$\mathtt{Adv}^{\mathsf{tda,eaa}}_{\Pi}(\mathcal{A}) = \Pr[\mathsf{guess}_0]$$

GAME 1. **[Exclude sessions without matching sessions]** In this game, we add an abort rule. A challenger aborts the game if a session does not have a matching session. Thus, we have:

$$\Pr[\mathsf{guess}_0] = \Pr[\mathsf{guess}_1] + \epsilon_{\mathsf{tda,eaa}}$$

GAME 2. **[Identify a session]** In this game, a challenger guesses a pair $(i^*, s^*) \xleftarrow{\$} [n_P] \times [n_S]$. The game proceeds as before but aborts if (i^*, s^*) is different from an adversary's chosen session (i, s). The probability that a challenger is correct is $\frac{1}{n_P n_S}$. Therefore, we get:

$$\Pr[\mathsf{guess}_1] = n_P n_S \cdot \Pr[\mathsf{guess}_2]$$

Note that due to GAME 1 where enforces a matching session, this game also identifies the peer of $\pi^{s^*}_{i^*}$. We will denote the peer $\pi^{t^*}_{j^*}$.

GAME 3. **[Replace the premaster secret]** In this game, we replace a uniform value from $\{0,1\}^n$ for the premaster secret established between $\pi^{s^*}_{i^*}$ and $\pi^{t^*}_{j^*}$, where n is the length of the premaster secret. Note that in TLS-DHE, the premaster secret is g^{ab} if $\pi^{s^*}_{i^*}$'s DH keypair is (a, g^a) and $\pi^{t^*}_{j^*}$'s DH keypair is (b, g^b). An adversary that can differentiate GAME 2 and GAME 3 can solve the DDH problem. Thus, we have:

$$\Pr[\mathsf{guess}_2] - \Pr[\mathsf{guess}_3] \leq \epsilon_{\mathsf{ddh}}$$

GAME 4. **[Replace the master secret]** In this game, we substitute a uniform value from $\{0,1\}^n$ for the master secret of $\pi^{s^*}_{i^*}$ and $\pi^{t^*}_{j^*}$, which is computed with a pseudorandom function PRF, where n is the length of the master secret.

Note that $ms = \texttt{PRF}(pms, label_{ms} \| r_C \| r_S)$ in GAME 3 and it is changed to $\tilde{ms} \xleftarrow{\$} \{0,1\}^n$ in GAME 4. Therefore, an adversary that can differentiate GAME 3 and GAME 4 can break PRF. Thus, we get:

$$\Pr[\mathsf{guess}_3] - \Pr[\mathsf{guess}_4] \leq \epsilon_{\mathsf{prf}}$$

GAME 5. **[Replace encryption keys]** In this game, we change encryption keys k of $\pi_{i*}^{s^*}$ and $\pi_{j*}^{t^*}$, which is computed as $k = \texttt{PRF}(ms, label_k \| r_C \| r_S)$, into a uniform value from $\{0,1\}^\ell$, where ℓ is the length of the encryption keys. As in the previous game, an adversary that can differentiate GAME 4 and GAME 5 can break PRF. Thus, we get:

$$\Pr[\mathsf{guess}_4] - \Pr[\mathsf{guess}_5] \leq \epsilon_{\mathsf{prf}}$$

Note that in this game, the encryption keys become independent of the key material since they are random values. This means that if an adversary can guess the correct answer in this game, he can break the stateful length hiding authenticated encryption. Therefore, $\Pr[\mathsf{guess}_5] = \epsilon_{\mathsf{slhae}}$

Taken all the above probabilities together, we get the result. □

4.6 Evaluation

We now evaluate the performance of TLS-SEED; we first discuss how we implement TLS-SEED and then present its performance.

4.6.1 SEED Implementation

SEED devices we first build two SEED devices based on Intel SGX and ARM TrustZone; the SGX-based SEED device is built on top of Intel NUC7CJYH with Intel Celeron J4005 at 2.00GHz and 8GB memory, while the ARM TrustZone-based SEED device is on top of Hikey960 with Cortex-A73 at 2.40GHz and 4GB memory.

We also implement (i) an ASP on Amazon Web Services (AWS) EC2 with Intel Xeon CPU E5-2676 at 2.40GHz and 1GB memory, and (ii) a client on a laptop equipped with Intel Core i7-7500U CPU at 2.70GHz and 8GB memory to request content.

TLS extensions for SEED We extend OpenSSL-1.1.1e to implement TLS-SEED: `TLS-FRONTEND-SEED` deployed on the client and the SEED device, and `TLS-BACKEND-SEED` on the SEED device and the service provider. We also implement other TLS extensions as a reference to ours;

Keyless SSL [98] and Delegated Credential [9][5]. The ciphersuites used in this experiment are `ECDHE-ECDSA-AES128-GCM-SHA256` for TLS 1.2 and `AEAD-AES128-GCM-SHA256` for TLS 1.3, which are widely used in practice [50]. All digests and signatures are generated based on the SHA-256 hash algorithm, which is a default hash function in TLS 1.2 [23].

4.6.2 Experiment Settings

We first locate the client and the SEED device in the same local network, and they communicate via WiFi. When locating an ASP, however, we need to vary the distance between the ASP and the SEED device to consider diverse cloud and edge settings; for example, a client may have to directly communicate with an ASP if unexpected errors occur on the SEED device or the application has not been loaded yet on the SEED device due to its incremental deployment.

Testbed	C- SEED (SGX)	SEED (SGX) - S	C - SEED (TZ)	SEED (TZ) - S
Intra-country	4.194ms	22.499ms	9.386ms	22.921ms
Inter-country	4.194ms	52.673ms	9.386ms	54.601ms

Table 4.2: **Networking Settings.** The table describes the round-trip times between entities in each scenario, where C and a SEED device (both SGX based and TrustZone (TZ) based) are in the same campus while S is located depending on the scenarios.

To this end, we first consider an experiment scenario where the ASP is located near the SEED device; we pick an EC2 instance whose region is located in the same country as the SEED device (called the *intra-country* scenario). We next find other EC2 instances located in the other regions; we deliberately choose one whose latency to the SEED device is at least twice larger than that of the instance located in the same country as the SEED device (called the *inter-country* scenario). Table 4.2 shows the network profiles of our settings: roundtrip times (100 trials of ping) between the client (C) and the SEED device, and the SEED device and the server (S) depending on the scenarios.

[5]Please refer to our implementation at `https://github.com/tls-seed`

4.6.3 Performance Evaluation

We evaluate the performance of the SEED platform from the perspective of the client and the edge device. First, the edge application is assumed to be located near the edge of the network; thus, its network latency to the client is small. However, all of the communications between the client and the edge application are now on top of the TLS protocol, thus making its initial handshake between the two parties heavily impact on users' quality of experience. To investigate its impact, we measure and analyze the handshake latency between the client and the edge device.

Second, the SEED device now has to spend extra CPU cycles for its cryptographic operations, such as generating signatures, which can impose more computational overhead than our expectation. Thus, we have to estimate the cost of TLS-SEED (i.e., CPU microbenchmark) to understand how much computation resource a single SEED device needs to efficiently scale out the number of edge applications.

Third, to benefit from TLS-SEED, the client should be upgraded to support TLS-FRONTEND-SEED, which is a hurdle in terms of deployability. Note that the SEED device can know whether the client supports TLS-FRONTEND-SEED when it receives a `ClientHello` message, SEED has a fallback mechanism for a client that does not support TLS-FRONTEND-SEED. To measure the fallback overhead, we evaluate the session establishment time between the client and the server with and without the SEED device. In the latter case, the SEED device is an application level relay.

Handshake Latency.

Figure 4.4 shows the experiment results and we make a number of observations. First, we observe that `Standard TLS` achieves the best performance in TLS handshakes in all scenarios, which is expected as the SEED device in `Standard TLS` just intercepts the TLS handshake destined to the server and pretends to be the server. Note that it does not provide any security benefits.

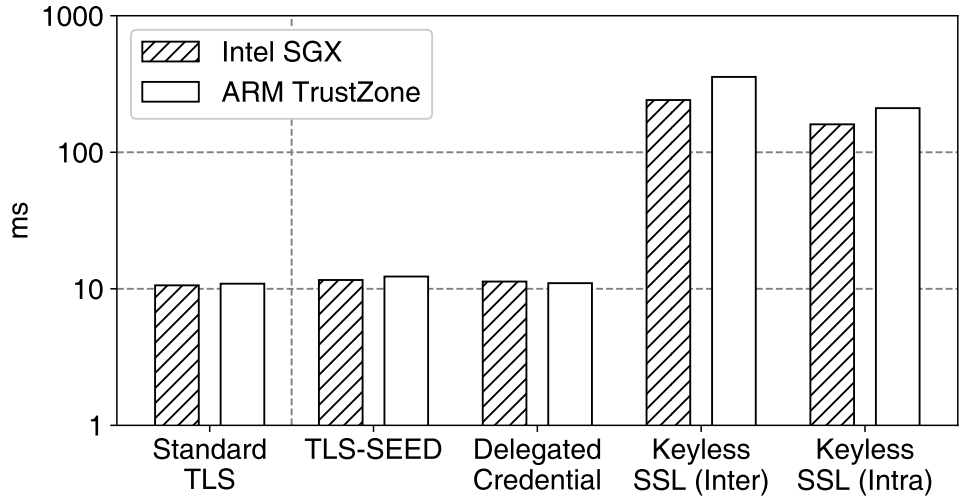


Figure 4.4: **Handshake Latency**

Second, we observe that `Keyless SSL` takes longer than the other schemes to finish the TLS handshake in all scenarios; this is because the SEED device in `Keyless SSL` needs to communicate with the server that performs private key operations even if the edge application services the client. This extra round trip incurs substantial delays when the server is located in the different region (160.18ms (intra-country) and 241.3ms (inter-country) in SGX, 210.46ms (intra-country) and 356.41ms (inter-country) in TrustZone), which shows 15.11 times (intra-country) and 22.76 times (inter-country) in the SGX scenario while 19.30 times (intra-country) and 32.70 times (inter-country) longer delay than that of `Standard TLS`.

Lastly, TLS-SEED shows the handshake delay of 11.6ms (in SGX) and 12.3 (in TrustZone), which incurs only 9.43% (in SGX) and 12.84% (in TrustZone) more overhead compared with `Standard TLS`, while meeting all the security goals that we specified. TLS-SEED also shows slightly more delay compared with `Delegated Credential`. The overhead is only 2.65% (in SGX) and 11.82% (in TrustZone). We break down this overhead and find out that the overhead is mainly caused when generating a signature over software attestation, which takes between 0.5ms and 1.2ms in the experiments.

Handshake CPU Microbenchmarks. We now evaluate the CPU processing time required to perform TLS handshakes on the SEED device. More specifically, we measure the elapsed time between the `ClientHello` to `Finished` message for each of the schemes in TLS 1.3.

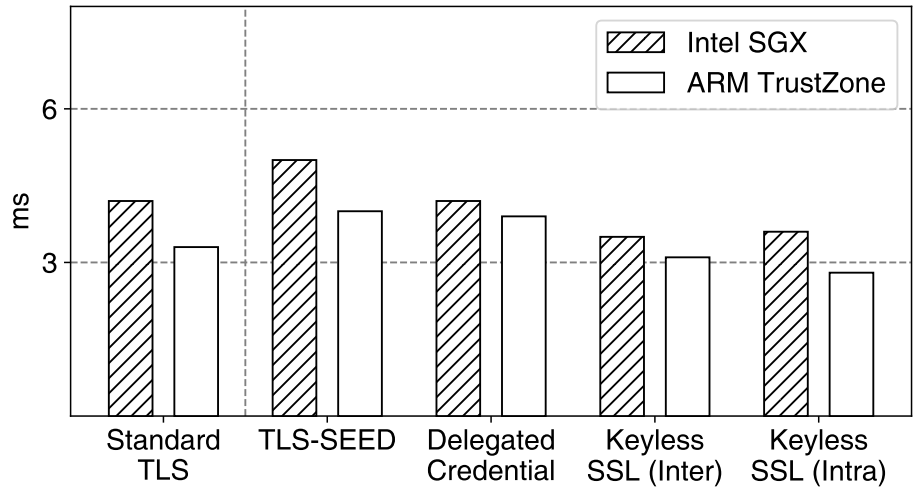


Figure 4.5: **Handshake CPU Microbenchmarks**

As Figure 4.5 shows, we find that TLS-SEED shows slightly more delay in the CPU processing time, compared with `Standard TLS` and `Delegated Credential`. The overhead is 5.80% (both `Standard TLS` and `Delegated Credential` in SGX), 8.33% (`Standard TLS` in TrustZone), and 5.41% (`Delegated Credential` in TrustZone) The result shows that SGX is slower than TrustZone. To understand this result, we execute the same program over the non-enclave and the normal world and find that the result is similar to the result over TEE. We conclude that the performance of the CPU accounts for the difference between SGX and TrustZone. Note that `Keyless SSL` delegates private key operations to key servers; thus, the CPU processing time in SEED devices are faster than those of other schemes.

Fallback Latency. When the client without TLS-SEED attempts to initiate a connection with the SEED edge device, the TLS connection is set up with the ASP. However, note that the SEED device still mediates as an application level relay, which might introduce additional

delay. If this incurs some overhead, it will not only exacerbate the performance of non-TLS-SEED clients, but also hinder the incremental SEED deployment. We now measure the time to establish a TLS session between the non-TLS-SEED client and the ASP *via the* SEED *device*, which is compared with the non-TLS-SEED client to directly initiate a TLS handshake with the ASP without any involvement of the SEED device.

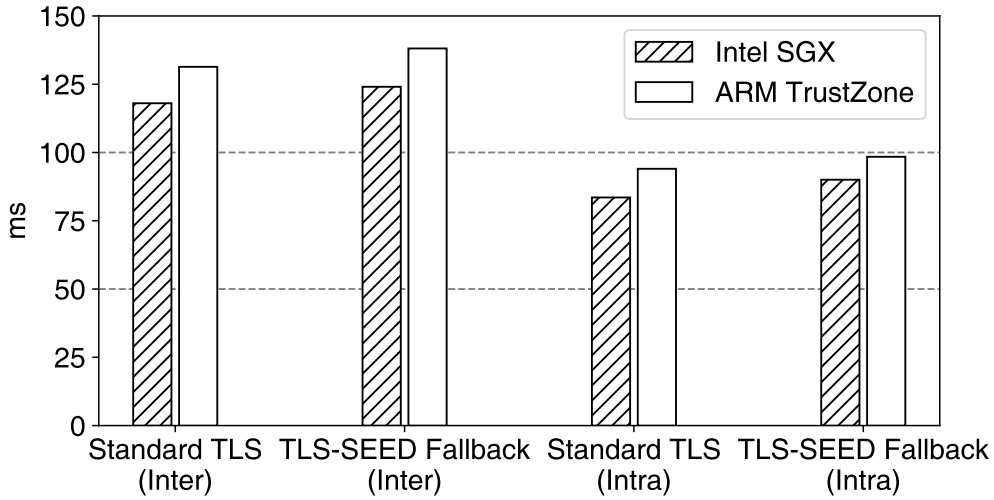


Figure 4.6: **Fallback Latency**

As shown in Figure 4.6, we notice that the overhead of the fallback mechanism is steady, but modest 5.91 ms more delays (5.68% overhead) on average. For example, when the SEED device is implemented based on TrustZone, it introduces additional 5.13% (intra-country) and 4.68% (inter-country) overhead on average. We believe this is not a significant performance penalty; thus, the SEED device is incrementally deployable, even with non-TLS-SEED clients.

Summary. From the experiments, we found that SEED introduces the negligible overhead in terms of TLS handshake delays and CPU cycles, while satisfying the security goals. We also observed that SEED introduces up to 5% of additional latency when the client is not SEED-compliant; however, we believe it is not a serious problem to non-TLS-SEED clients and not a hurdle to deploying the SEED platform.

4.7 Discussions

4.7.1 Incremental Deployment Scenario

It is worth noting that the SEED platform requires modification onto both ASPs and clients, which can be a major obstacle for a greater deployment. However, on a positive note, we believe that, with marginal actions by relatively few parties, transitioning to the SEED platform is not the illusory goal: A *client* needs to support only the TLS-FRONTEND-SEED to establish the session with a SEED device, of which the additional requirement is limited to adding cryptographic libraries An *ASP* needs only to deploy a CC server that performs the private key operation to generate CC, which does not have to be integrated (thus, require modification) to the existing platform.

4.7.2 Mobility Support

As in §3.7.5, it is necessary to consider how TLS-SEED can support mobility. The terminology used in discussion follows [62]. Since edge applications are generally loaded at a gateway or a base station, the connection would be disconnected when a mobile host leave the corresponding network. To maintain a session, a migration should be proceeded from one gateway (or a base station) to another one. Then, a mobile host can be served with a migrated context or application.

4.7.3 Dependency on TEEs

Note that CC has dependency on TEEs since different TEEs guarantee different properties. For example, Intel SGX provides an instruction to measure a particular enclave, while it is not true over TrustZone. Furthermore, Intel SGX is independent of the secure booting; thus, it is sufficient for a CC to include the hash value of the enclave. However, TrustZone aims to perform the secure booting from a bootloader to a trusted application. Therefore, if we wants to make CC to provide sufficient information to verify a TrustZone device precisely,

CC should include all the hash value of the software stack including a bootloader, a trusted operating system, and a trusted application.

4.8 Conclusion

With an increasing number of connected devices and the popularity of delay sensitive-services such as autonomous vehicles and virtual reality, edge computing has gained momentum. Although many proposals have been introduced for edge computing platforms leveraging virtualization technologies, the security technologies on the edge computing platforms are not mature yet. Since the edge computing platform is usually installed and managed by a third party, it should have a mechanism that each communication party can trust and verify the platform and its applications.

To achieve this goal, we presented the SEED platform leveraging the TEE, which is accountable to ASPs and visible to clients due to TLS-SEED. One of the central elements of TLS-SEED is a Cross Credential (CC) that represents a trust relation between an ASP and an edge computing platform. A CC not only frees the ASP from sharing his private key, but also allows clients to attest edge applications. Moreover, the management overhead of CCs on the SEED platform is marginal since the architecture requires SEED to protect only its private key, which significantly reduces the attack surface. We also formally proved that TLS-SEED by introducing the ACCE-SEED model where three participants are involved in a session. Also, we implemented SEED on the Intel SGX platform (Intel NUC NUC7CJYH) and the ARM TrustZone platform (Hikey960 with OP-TEE), which are two most widely deployed TEE technologies and demonstrated the TLS-SEED extensions by using the OpenSSL-1.1.1e library. From the experiments, we showed that the security of the edge computing platform could be achieved without introducing substantial communication and computation overheads.

Chapter 5

Conclusion

Recently, a middlebox needs a method to be participate into encrypted sessions, as HTTPS becomes a *de facto* standard protocol on Internet. The widely used TLS interception scheme, so-called SplitTLS, is knows with various security vulnerabilities; thus, many require a protocol to make middleboxes securely participate into TLS sessions.

To this end, we classify a middlebox into two types – a middlebox-as-a-middlebox and a middlebox-as-an-endpoint, analyze 23 previous studies (14 for a middlebox-as-a-middlebox and 9 for a middlebox-as-an-endpoint), and get lessons respectively. Based on the learnings, we propose MATLS for a middlebox-as-a-middlebox and TLS-SEED for a middlebox-as-an-endpoint (especially, considering edge computing). We present MATLS considering seven security properties that we propose to secure a TLS session with middleboxes and we prove all the properties are achieved during the execution of the MATLS protocol by using a state-of-the-art security verification tool Tamarin. MATLS shows marginal overheads compared with SplitTLS in our testbed-based experiments. TLS-SEED provides an application service provider and a client with a way to securely communicate with each other in edge computing platforms. It allows an application service provider not to share his private keys with platforms, while it helps a client to perform remote attestation of an edge application without relying on attestation services. The TLS-FRONTEND-SEED protocol demonstrates that it can achieve its properties only with negligible overheads compared with SplitTLS in edge computing

platforms.

We leave two challenges for future work.

Middlebox Reputation. Note that many vulnerabilities from SplitTLS are from a incorrectly implemented middlebox or a mis-configured middlebox. To reduce the risk from incorrct middleboxes, there needs not only a protocol that audits a middlebox like MATLS, but also a system that manages reputation of middleboxes to allow endpoints to decide whether to opt-in a middlebox in the TLS session. With MATLS, endpoints can report incorrect middleboxes to a management system and the system can provide endpoints with information about middleboxes.

User Notification and Authorization. As [83] indicates that many people at least wants to know whether the third-party middleboxes inspect his/her traffic. Note that there is no information about middleboxes to users in practice. The protocols that assume a middlebox has its own certificate can provide a user with information about a middlebox (e.g., a name of a middlebox), but a user cannot know whether to accept a middlebox. Therefore, we should specify what information should be provided to a user to decide whether to authorize a middlebox and consider a convenient user interface for a user to (de)authorize a middlebox with sufficient information about a middlebox.

Bibliography

[1] E. 3rd Generation Partnership Project (3GPP), "5g; security architecture and procedures for 5g system (3gpp ts 33.501 version 15.1.0 release 15)," https://www.etsi.org/deliver/etsi_ts/133500_133599/133501/15.01.00_60/ts_133501v150100p.pdf, 2018, accessed: 2019-10-20.

[2] M. access Edge Computing (MEC) ETSI Industry Specification Group (ISG), "Multi-access edge computing (mec); phase 2: Use cases and requirements," https://www.etsi.org/deliver/etsi_gs/MEC/001_099/002/02.01.01_60/gs_MEC002v020101p.pdf, 2018, accessed: 2019-10-20.

[3] 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.

[4] H. Ahlehagh and S. Dey, "Video-aware scheduling and caching in the radio access network," *IEEE/ACM Transactions on Networking (TON)*, vol. 22, no. 5, pp. 1444–1462, October 2014.

[5] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13. ACM New York, NY, USA, 2013.

[6] ARM, "ARM security technology building a secure system using TrustZone technology (white paper)," *ARM Limited*, 2009.

[7] E. Asanghanwa, "Simplifying confidential computing: Azure iot edge security with enclaves – public preview," https://azure.microsoft.com/en-us/blog/simplifying-confidential-computing-azure-iot-edge-security-with-enclaves-public-preview/, 2018, accessed: 2020-04-21.

[8] W. Ashford, "Privdog ssl compromise potentially worse than superfish," 2015, accessed: 2020-06-05. [Online]. Available: https://www.computerweekly.com/news/2240241126/PrivDog-SSL-compromise-potentially-worse-than-Superfish

[9] R. Barnes, S. Iyengar, N. Sullivan, and E. Rescorla, "Delegated credential for TLS (RFC draft)," February 2019. [Online]. Available: https://tools.ietf.org/pdf/draft-ietf-tls-subcerts-03.pdf

[10] 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.

[11] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using AirBox," in *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 2016, pp. 14–27.

[12] K. Bhardwaj, M.-W. Shih, A. Gavrilovska, T. Kim, and C. Song, "Spx: Preserving end-to-end security for edge computing," 2018.

[13] K. Bhargavan, I. Boureanu, P.-A. Fouque, C. Onete, and B. Richard, "Content delivery over TLS: a cryptographic analysis of keyless SSL," in *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*. IEEE, April 2017, pp. 1–16.

[14] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[15] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, "Post-quantum key exchange for the tls protocol from the ring learning with errors problem," in *Security and Privacy (S&P), 2015 IEEE Symposium on*. IEEE, 2015, pp. 553–570.

[16] 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.

[17] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 41, no. 1. ACM, 2013.

[18] 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.

[19] Comodo, "Comodo report of incident - comodo detected and thwarted an intrusion on 26-mar-2011," 2011, accessed: 2020-06-09. [Online]. Available: https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html

[20] E. D. Migault, "LURK protocol for TLS/DTLS1.2 version 1.0 (RFC draft)," 2017.

[21] ——, "Lurk extension version 1 for tls 1.3 authentication (rfc draft)," 2018. [Online]. Available: https://tools.ietf.org/html/draft-mglt-lurk-lurk-00

[22] 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.

[23] T. Dierks, "The Transport Layer Security (TLS) protocol version 1.2," 2008.

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

[25] 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.

[26] G. Eriksson, J. Mattsson, N. Mitra, and Z. Sarker, "Blind cache: a solution to content delivery challenges in an all-encrypted web," *Ericsson review*, 2017.

[27] EU, "General data protection regulation article 5." 2018. [Online]. Available: https://gdpr-info.eu/art-5-gdpr/

[28] Facebook, "Introducing our certificate transparency monitoring tool," accessed: 2020-06-09. [Online]. Available: https://www.facebook.com/notes/protect-the-graph/introducing-our-certificate-transparency-monitoring-tool/1811919779048165/

[29] 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*, 2017, pp. 1323–1338.

[30] T. Fossati, V. K. Gurbani, and V. Kolesnikov, "Love all, trust few: On trusting intermediaries in http," in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015, pp. 1–6.

[31] Gaurang, "Nokia's mitm on https traffic from their phone," 2013. [Online]. Available: https://gaurangkp.wordpress.com/2013/01/09/nokia-https-mitm/

[32] Google, "Https encryption on the web," https://transparencyreport.google.com/https/overview, accessed: 2019-09-02.

[33] O. C. A. W. Group *et al.*, "Openfog reference architecture for fog computing," https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf, p. 162, 2017, accessed: 2019-10-20.

[34] A. Guzman, K. Nekritz, and S. lyengar, "Delegated credentials for tls," https://blog.cloudflare.com/keyless-delegation/, 2019, accessed: 2020-04-29.

[35] 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.

[36] P. Hoffman and J. Schlyter, "The DNS-based Authentication of Named Entities (DANE) Transport Layer Security (TLS) protocol: TLSA," IETF, RFC 6698, 2012.

[37] A. Holst, "Security as a service revenues worldwide 2018-2024," 2020, accessed: 2020-05-29. [Online]. Available: https://www.statista.com/statistics/595164/worldwide-security-as-a-service-market-size/

[38] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 public key infrastructure certificate and CRL profile," IETF, RFC 5280, 1998.

[39] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 533–549.

[40] Intel, "Edge computing – intel," https://www.intel.com/content/www/us/en/edge-computing/overview.html, accessed: 2020-05-05.

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

[42] K. Jacobs, J. Jones, and T. van der Merwe, "Validating delegated credentials for tls in firefox," https://blog.mozilla.org/security/2019/11/01/validating-delegated-credentials-for-tls-in-firefox/, accessed: 2020-04-29.

[43] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of tls-dhe in the standard model," in *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 273–293.

[44] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "Privatezone: Providing a private execution environment using arm trustzone," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 797–810, 2018.

[45] J. Jarmoc and D. Unit, "SSL/TLS interception proxies and transitive trust," in *Black Hat Europe*, 2012.

[46] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel® software guard extensions: Epid provisioning and attestation services," pp. 1–10, 2016.

[47] S. Kekki, W. Featherstone, Y. Fang, P. Kuure, A. Li, A. Ranjan, D. Purkayastha, F. Jiangping, D. Frydman, G. Verin *et al.*, "Mec in 5g networks," in *ETSI white paper*, 2018, no. 28.

[48] S. Kent and R. Atkinson, "Security architecture for the internet protocol," 1998.

[49] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[50] P. Kotzias, A. Razaghpanah, J. Amann, K. G. Paterson, N. Vallina-Rodriguez, and J. Caballero, "Coming of age: A longitudinal study of tls deployment," in *Proceedings of the Internet Measurement Conference 2018*. ACM, 2018, pp. 415–428.

[51] 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.

[52] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency," 2013.

[53] A. Levy, H. Corrigan-Gibbs, and D. Boneh, "Stickler: Defending against malicious content distribution networks in an unmodified browser," *IEEE Security & Privacy*, vol. 14, no. 2, pp. 22–28, 2016.

[54] J. Li, R. Chen, J. Su, X. Huang, and X. Wang, "Me-tls: Middlebox-enhanced tls for internet-of-things devices," *IEEE Internet of Things Journal*, 2019.

[55] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 75–88.

[56] J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu, "When HTTPS meets CDN: A case of authentication in delegated service," in *Security and Privacy (S&P), 2014 IEEE symposium on*. IEEE, 2014, pp. 67–82.

[57] C. Liu, Y. Cui, K. Tan, Q. Fan, K. Ren, and J. Wu, "Building generic scalable middlebox services over encrypted protocols," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 2195–2203.

[58] P. Liu, D. Willis, and S. Banerjee, "ParaDrop: Enabling lightweight multi-tenancy at the network's extreme edge," in *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 2016, pp. 1–13.

[59] P. G. Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, 2015.

[60] 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

[61] G. Ma, Z. Wang, M. Zhang, J. Ye, M. Chen, and W. Zhu, "Understanding performance of edge content caching for mobile video streaming," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 5, pp. 1076–1089, 2017.

[62] J. Manner, M. Kojo, T. Suihko, P. Eardley, and D. Wisely, "Mobility related terminology," 2004.

[63] 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

[64] 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.

[65] MENAFN, "Network security appliance market 2019 global analysis, segments, size, share, industry growth and recent trends by forecast to 2023," 2019, accessed: 2020-05-29. [Online]. Available: https://menafn.com/1098037674/India-Network-Security-Appliance-Market-2019-Global-Analysis-Segments-Size-Share-Industry-Growth-and-Recent-Trends-by-Forecast-to-2023

[66] 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/

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

[68] 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.

[69] 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.

[70] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. 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.

[71] 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

[72] D. O'Brien, "Certificate transparency enforcement in google chrome," 2018, accessed: 2020-06-09. [Online]. Available: https://groups.google.com/a/chromium.org/forum/#!msg/ct-policy/wHILiYf31DE/iMFmpMEkAQAJ

[73] 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.

[74] M. Patel, B. Naughton, C. Chan, N. Sprecher, S. Abeta, A. Neal *et al.*, "Mobile-edge computing introductory technical white paper," pp. 1089–7801, 2014.

[75] K. G. Paterson, T. Ristenpart, and T. Shrimpton, "Tag size does matter: Attacks and proofs for the tls record protocol," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2011, pp. 372–389.

[76] R. Peon, "Explicit proxies for http/2.0," 2012.

[77] S. K. Platform, "Knox white paper – root of trust." [Online]. Available: https://docs.samsungknox.com/whitepapers/knox-platform/hardware-backed-root-of-trust.htm

[78] 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.

[79] A. Popov, "Prohibiting rc4 cipher suites," p. 25, 2015.

[80] J. Reschke and S. Loreto, "'out-of-band' content coding for HTTP (RFC draft)," 2017.

[81] E. Rescorla, "Http over tls," 2000. [Online]. Available: https://tools.ietf.org/pdf/rfc2818.pdf

[82] ——, "The Transport Layer Security (TLS) protocol version 1.3," 2018.

[83] S. Ruoti, M. O'Neill, D. Zappala, and K. E. Seamons, "User attitudes toward the inspection of encrypted traffic." in *SOUPS*, 2016, pp. 131–146.

[84] D. Sabella, V. Sukhomlinov, L. Trang, S. Kekki, P. Paglierani, R. Rossbach, X. Li, Y. Fang, D. Druta, F. Giust *et al.*, "Developing software for multi-access edge computing," 2019.

[85] K. Sasaki, N. Suzuki, S. Makido, and A. Nakao, "Vehicle control system coordinated between cloud and mobile edge computing," in *2016 55th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, Sep. 2016, pp. 1122–1127.

[86] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[87] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, 2009.

[88] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, "Supporting third party attestation for intel® sgx with intel® data center attestation primitives," 2018.

[89] M. Schneider, J. Rambach, and D. Stricker, "Augmented reality based on edge computing using the example of remote live support," in *2017 IEEE International Conference on Industrial Technology (ICIT)*, March 2017, pp. 1277–1282.

[90] 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/

[91] 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," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[92] 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.

[93] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[94] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs." *IACR Cryptology ePrint Archive*, vol. 2004, p. 332, 2004.

[95] S. Slovetskiy, "Approaches to HTTPS-based request routing and delegation (RFC draft)," 2015.

[96] SSLMate, "Cert spotter," accessed: 2020-06-09. [Online]. Available: https://sslmate.com/certspotter/

[97] StarlingX, "Starlingx," 2018. [Online]. Available: https://starlingx.io

[98] D. Stebila and N. Sullivan, "An analysis of tls handshake proxying," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 1. IEEE, 2015, pp. 279–286.

[99] N. Sullivan and W. Ladd, "Delegated credentials: Improving the security of tls certificates," https://engineering.fb.com/security/delegated-credentials/, accessed: 2020-04-29.

[100] C. S. B. Y. L. A. A. N. TELEFONICA S.A., Cadzow Communications, "Cyper; middlebox security protocol; part2: Transport layer msp, profile for fine grained access control," 2018, accessed: 2020-06-11. [Online]. Available: https://docplayer.net/88122390-Announcement-of-middlebox-security-protocol-msp-draft-parts.html

[101] 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.

[102] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 991–1008.

[103] 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.

[104] X. Wang, M. Chen, T. Taleb, A. Ksentini, and V. Leung, "Cache in the air: exploiting content caching and delivery techniques for 5G systems," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 131–139, 2014, https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-small-cell-study.pdf (Retrieved July 12, 2018).

[105] X. Wang, T. Taleb, Z. Han, S. Xu, and V. C. Leung, "Content-centric collaborative edge caching in 5G mobile internet," *IEEE Wireless Communications*, vol. 25, no. 3, 2018.

[106] 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/.

[107] J. Wilson, R. S. Wahby, H. Corrigan-Gibbs, D. Boneh, P. Levis, and K. Winstein, "Trust but verify: Auditing the secure internet of things," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 464–474.

[108] L. Zhu, N. Williams, and J. Altman, "Channel bindings for tls," https://tools.ietf.org/html/rfc5929, 2010, accessed: 2019-10-15.

Appendix A

Cryptographic Definitions

A.1 Cryptographic Definitions

In this section, we first introduce a formal model, called ACCE-SEED, that is a modified version of the authenticated and confidential channel establishment (ACCE) security model [43]. The model consists of two phases called a pre-accept phase (corresponding to the TLS handshake protocol) and a post-accept phase (corresponding to the TLS record protocol), the latter of which is based on the stateful length-hiding authenticated encryption, proposed by Paterson et al. [75]. Since there are three participants involved in our scenario, the model should be modified by introducing a SEED device as a new participant. Therefore, we first describe our modified model called ACCE-SEED and then define the lemmas, followed by the proof based on sequence of games [94]. We reuse the notation in [15, 13].

DEFINITION 4 **(Digital signature scheme)** A digital signature scheme $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ is defined as follows:

- $\mathsf{Gen}(1^\lambda) \xrightarrow{\$} (sk, pk)$: A probabilistic key generation function that takes a security parameter λ as input and outputs a signature key sk and a verification key pk.
- $\mathsf{Sign}(sk, m) \xrightarrow{\$} \sigma$: A probabilistic signing algorithm that takes a signing key sk and a message $m \in \{0,1\}^*$ as input and outputs a corresponding signature σ.

- $\mathsf{Vrfy}(pk, m, \sigma) \to \{0, 1\}$: A deterministic verification algorithm that takes a verification key pk, message m, and the corresponding signature σ. It outputs 0 (verification failure) or 1 (verification success).

The security of the digital signature scheme is defined with the adversary's advantage in the experiment of the existential unforgeability under chosen message attack (euf-cma), described as:

$$\begin{aligned}\mathtt{Adv}_{\Pi}^{\mathsf{euf\text{-}cma}}(\mathcal{A}) =& \Pr(\Pi.\mathsf{Vrfy}(pk, m^*, \sigma^*) = 1 : \\ & (sk, pk) \xleftarrow{\$} \Pi.\mathsf{Gen}(1^\lambda); \\ & (m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\Pi.\mathsf{Sign}(sk,\cdot)}(pk)),\end{aligned}$$

where the security parameter λ is given, m^* is a challenge message, and σ^* is an output of the adversary. The adversary $\mathcal{A}$ takes a verification key pk and interacts with an oracle $\Pi.\mathrm{Sign}(sk, \cdot)$, but do not query m^* to the oracle.

We say a digital signature scheme is euf-cma secure if there exists a negligible function $\epsilon_{\mathsf{euf\text{-}cma}}$ such that:

$$\mathtt{Adv}_{\Pi}^{\mathsf{euf\text{-}cma}}(\mathcal{A}) \leq \epsilon_{\mathsf{euf\text{-}cma}}$$

DEFINITION 5 **(Secure hash function)** A secure (keyed) hash function $\Pi = (\mathsf{Gen}, H)$ that guarantees collision resistance is defined as follows:

- $\mathsf{Gen}(1^\lambda) \xrightarrow{\$} s$: A probabilistic key generation function that takes a security parameter λ as input and outputs a key s.
- H: A deterministic algorithm that takes a key s and a message $m \in \{0, 1\}^*$ as input and outputs a string $H^s(x) \in \{0, 1\}^\lambda$.

The security of the secure hash function is defined with the adversary's advantage in the

collision attack (denoted by coll), described as:

$$\begin{aligned}\mathtt{Adv}^{\mathsf{coll}}_{\Pi}(\mathcal{A}) =& \Pr(\Pi.H^s(m) = \Pi.H^s(m') \wedge m \neq m' : \\ & (s) \xleftarrow{\$} \Pi.\mathsf{Gen}(1^\lambda); \\ & m, m' \in \{0,1\}^*).\end{aligned}$$

where the security parameter λ is given. We say a hash algorithm is collision (coll) resistant if there exists a negligible function ϵ_{coll} such that:

$$\mathtt{Adv}^{\mathsf{coll}}_{\Pi}(\mathcal{A}) \leq \epsilon_{\mathsf{coll}}$$

DEFINITION 6 **(Pseudo random function)** A pseudo random function is a deterministic function $\mathsf{PRF}(k, x)$ that takes a key $k \in \{0,1\}^{\lambda_1}$ and a message $x \in \{0,1\}^*$ and outputs a message $z \in \{0,1\}^{\lambda_2}$.

The security of a pseudo random function is defined with the adversary's advantage in the distinguishing messages from a random world and a pseudo random world, with a restriction that $\mathcal{A}$ has never issued $\mathsf{PRF}(k, \cdot)$ queries.

That is:

$$\begin{aligned}\mathtt{Adv}^{\mathsf{prf}}_{\Pi}(\mathcal{A}) =& \Pr(b = b' : k \xleftarrow{\$} \{0,1\}^{\lambda_1}; x \xleftarrow{\$} \mathcal{A}^{F(k,\cdot)}(); \\ & k_0 \leftarrow \mathsf{PRF}(k, x); k_1 \xleftarrow{\$} \{0,1\}^{\lambda_2}; \\ & b \xleftarrow{\$} \{0,1\}; b' \xleftarrow{\$} \mathcal{A}^{F(k,\cdot)}(k_b)).\end{aligned}$$

where the security parameters λ_1, λ_2 are given. We say a pseudo random function is prf-secure if there exists a negligible function ϵ_{prf} such that:

$$\mathtt{Adv}^{\mathsf{prf}}_{\Pi}(\mathcal{A}) \leq \epsilon_{\mathsf{prf}}$$

DEFINITION 7 **(Stateful length-hiding authenticated encryption)** A stateful length-hiding authenticated encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Init}, \mathsf{Enc})$ is defined as follows:

- $\mathsf{Gen}(1^\lambda) \xrightarrow{\$} k$: A probabilistic key generation function that takes a security parameter λ and outputs a key $\in \{0,1\}^\lambda$.

- $\mathsf{Init}() \rightarrow (st_E, st_D)$: A deterministic function that outputs initial encryption and decryption states, st_E and st_D.

- $\mathsf{Enc}(k, \ell, ad, m, st_E) \xrightarrow{\$} (c, st'_E)$: A probabilistic encryption algorithm that takes a key k, length $\ell \in \mathbb{N}$, associated data $ad \in \{0,1\}^*$, message m, and an encryption state st_E as input and outputs a ciphertext c or $\perp$ with an updated encryption state st'_E), where $|c| = \ell$ if $c \neq \perp$.

- $\mathsf{Dec}(k, ad, c, st_D) :\rightarrow (m, st'_D)$ A deterministic decryption algorithm that takes a key k, associated data ad, a ciphertext c, and a decryption state st_D as input and outputs a decrypted message m with an updated decryption state st'_D.

The security of the stateful length-hiding authenticated encryption is defined with the adversary's advantage as follows:

$$\begin{aligned}\mathtt{Adv}^{\mathsf{slhae}}_{\Pi}(\mathcal{A}) =& \Pr(b = b' : k \xleftarrow{\$} \Pi.\mathsf{Gen}(\lambda); (st_E, st_D) \leftarrow \Pi.\mathsf{Init}();\\ & b \xleftarrow{\$} \{0,1\}; b' \xleftarrow{\$} \mathcal{A}^{\mathsf{Enc},\mathsf{Dec}}()),\end{aligned}$$

where Enc and Dec denote an encryption oracle and a decryption oracle whose algorithms are described in §A.2. Note that they have their own internal state value and increased by one whenever they are queried. We say a stateful length-hiding authenticated encryption is slhae-secure if there exists a negligible function $\epsilon_{\mathsf{slhae}}$ such that:

$$\mathtt{Adv}^{\mathsf{slhae}}_{\Pi}(\mathcal{A}) \leq \epsilon_{\mathsf{slhae}}$$

DEFINITION 8 **(Decisional Diffie-Hellman Assumption)** Let G be a group of prime order q and g be a generator of G. Given (g, g^a, g^b, g^c) for $a, b, c \in \mathbb{Z}_q$, the decisional Diffie-Hellman (DDH) assumption means that there exists a negiligible function ϵ_{ddh} such that:

$$|\Pr[\mathcal{A}(g, g^a, g^b, g^{ab}) = 1] - \Pr[\mathcal{A}(g, g^a, g^b, g^c) = 1]| \leq \epsilon_{\mathsf{ddh}}$$

A.2 Oracles

In the ACCE and ACCE-SEED models, the adversary uses an encryption/decryption oracle. Both oracles are described in Algorithm 1 and Algorithm 2 below. They manage states of a particular session and update the states after encryption or decryption.

Algorithm 1 $\texttt{Enc}(i, s, \ell, ad, m_0, m_1)$

[1] $\pi_i^s.u = \pi_i^s.u + 1$ $(C^0, \pi_i^s.st_E^0) \xleftarrow{\$} \texttt{AEAD.Enc}(\pi_i^s.k, \ell, ad, m_0, \pi_i^s.st_E)$ $(C^1, \pi_i^s.st_E^1) \xleftarrow{\$} \texttt{AEAD.Enc}(\pi_i^s.k, \ell, ad, m_1, \pi_i^s.st_E)$ **if** $C^0 = \perp$ or $C^1 = \perp$ **then return** $\perp$ $\pi_i^s.C_{\pi_i^s.u} \leftarrow C^{\pi_i^s.b}$, $\pi_i^s.ad_{\pi_i^s.u} \leftarrow ad$, $\pi_i^s.st_E \leftarrow st_E^{\pi_i^s.b}$ **return** $C^{\pi_i^s.b}$

Algorithm 2 $\texttt{Dec}(i, s, ad, C)$

[1] **if** $\pi_i^s.b = 0$ **then return** $\perp$ π_j^t is the matching session at P_j with π_i^s $\pi_i^s.v \leftarrow \pi_i^s.v + 1$ $(m', \pi_i^s, st_D \leftarrow \texttt{AEAD.Dec}(\pi_i^s.k, ad, C, \pi_i^s.st_D)$ **if** $\pi_i^s.v > \pi_j^t.u$ or $c \neq \pi_j^t.C_{\pi_i^s.v}$ or $ad \neq \pi_j^t.ad_{\pi_i^s.v}$ **then** $\pi_i^s.phase \leftarrow 1$ **if** $\pi_i^s.phase = 1$ **then return** m **return** $\perp$

국문초록

인터넷 트래픽이 HTTPS로 암호화되면서 웹 캐시나 방화벽 같은 미들박스는 특별한 조치가 없이는 동작하기 어려운 상태가 되었다. 그러다보니 현업에서는 미들박스를 암호화된 세션에서 활용하기 위해 공개키 인증 구조의 신뢰 방식을 오용하여 SplitTLS 라고 하는 TLS를 가로채는 기법을 사용하고 있다. 그렇지만 지난 몇 년간 발표된 여러 논문에서는 미들박스가 잘못 구현되었거나 미들박스 설정이 잘못되어 SplitTLS 를 수행하는데 있어 여러 보안 문제가 발생하고 있다는 것이 밝혀졌다.

이 논문은 미들박스가 TLS 세션에 안전하고 신뢰성있게 참여하기 위한 방법을 설계하고자 한다. 이를 위해 우리는 먼저 미들박스를 중간자 역할을 수행하는 미들박스와 종단점 역할을 수행하는 미들박스로 구분하였다. 중간자 역할을 수행하는 미들박스란 통신 시간에 서버와 클라이언트 가운데서 동작을 수행하는 중개자이며, 종단점 역할을 수행하는 미들박스란 세션이 활성화된 동안 미들박스가 서버처럼 동작하는 개체를 의미한다. 전자의 예로는 침입 탐지 시스템이 있으며 후자의 예로는 웹 캐시가 있다. 이 구분 하에서 우리는 미들박스를 TLS 세션에 참여시키기 위한 23개의 프로토콜에 대해 검토하였다. 23개 중 14개는 중간자 역할을 수행하는 미들박스를 위한 프로토콜이며, 9개는 종단점 역할을 수행하는 미들박스를 위한 프로토콜이다.

우리는 선행 연구를 검토하면서 다음의 교훈을 얻었다. 우선 중간자 역할을 수행하는 미들박스를 위한 프로토콜을 설계하는데 있어서 먼저 고려해야 할 점은, 미들박스가 과도한 퍼미션을 갖지 않도록 최소 권한을 줄 수 있는 방법을 찾아야 한다는 것이었다. 또한, 서버가 세션에 참여하기 때문에, 서버가 암호학적 방법을 통해서 클라이언트에게 미들박스에 대한 정보를 줄 수 있다는 점도 고려할 수 있다는 것을 알게 되었다. 다음으로 종단점 역할을 수행하는 미들박스를 위한 프로토콜을 설계하는데 있어서 중요하게 고려해야 할 점은 서버가 세션에 참여하지 않기 때문에, 서버로의 통신이 추가되는 것은 바람직하지 않다는 점과 키 관리에 있어서 서버에게 부하가 가지 않도록 되어야 하며 기밀 키의 개수는

최소화할 수 있어야 한다는 점이었다.

이 논문에서는 위 교훈점을 바탕으로 maTLS 와 TLS-Seed 라는 두 개의 프로토콜을 제안하였다.

먼저, maTLS 프로토콜은 중간자 역할을 수행하는 미들박스를 위한 프로토콜이다. 현재 미들박스를 보안 세션에 참여시키기 위한 SplitTLS 라는 프로토콜은 매우 많은 보안 문제점이 발견 되었다. 여러 선행 연구들이 TLS와 미들박스를 결합하면서 인증서 검증 실패나 오래된 암호 기법을 사용하거나 원치 않는 수정을 한다는 것을 밝혀 내었다. 이러한 보안 취약점을 해결하기 위해 우리는 maTLS 프로토콜을 제안하였다. 이 프로토콜은 미들박스가 TLS 세션에 자신을 드러내면서 감독될 수 있는 형태로 참여하도록 한다. TLS 세션에 참여하는 모든 미들박스들은 세션을 두 개의 세그먼트로 분할하며 각 세그먼트는 해당 세그먼트를 위한 보안 파라미터를 갖는다. maTLS 프로토콜은 미들박스를 인증하고 각 세그먼트들의 보안 파라미터를 검증하며, 미들박스의 쓰기 연산을 감독하도록 설계되었다. 이렇게 하여 전체 세션의 보안성이 보장된다. 이 보안성이 실제 달성된다는 것을 보이기 위해 우리는 최신 보안성 검증 도구인 Tamarin을 활용하여 증명하였으며 실제 테스트베드 실험을 통해 maTLS 가 약간의 오버헤드를 가지면서 위 보안성 목표를 달성한다는 것을 보였다.

다음으로 TLS-Seed 는 종단점 역할을 수행하는 미들박스를 위한 프로토콜이다. 특별히 우리는 엣지 컴퓨팅 시나리오를 고려하면서 이 프로토콜을 설계하였다. 엣지 컴퓨팅이란 계산과 저장 노드를 클라이언트에 가깝게 위치시켜서 클라이언트에게는 빠른 응답을 제공하고 서버에게는 대역폭 부하를 줄이도록 한다. 일반적으로 엣지 컴퓨팅 플랫폼은 애플리케이션 제공자나 클라이언트에게 서드 파티이기 때문에 이 두 개체는 모두 높은 수준의 보안성을 요구할 것이다. 이에 따라 우리는 TLS-Seed 를 제안하였으며, 이를 통해 위험한 개인키 공유 문제와 비효율적인 원격 입증 문제를 해결하고자 하였으며, 동시에 엣지 컴퓨팅의 장점인 성능 향상을 유지토록 하고자 하였다. TLS-Seed 는 애플리케이션 서비스 제공자가 i) 자신의 개인키를 공유하지 않으면서도 엣지 애플리케이션을 도입할 수 있도록 만들어주고, ii) 원격 입증을 수행하여 엣지 애플리케이션을 인가하거나 비인가할 수 있도록 해준다. 또한 애플리케이션 서비스 제공자가 클라이언트에게 엣지 애플리케이

션에 대한 충분한 정보를 제공하여 클라이언트가 입증 서비스에 의존하지 않더라도 엣지 애플리케이션에 대해 이해할 수 있도록 해준다. TLS-SEED 를 위한 핵심 자료 구조는 CROSSCREDENTIAL (CC)이며, 이는 클라이언트에게 애플리케이션 제공자와 신뢰할 수 있는 기기 사이의 신뢰 관계를 명시적으로 보여준다. CC 는 또한 클라이언트가 엣지 애플리케이션의 무결성을 검증할 수 있도록 충분한 정보를 제공한다. TLS-SEED 를 수학적으로 증명하기 위해, 우리는 ACCE-SEED 라는 TLS-SEED 를 위한 보안 모델을 도입하였다. 이 모델은 TLS를 위한 ACCE 모델을 TLS-SEED 에 적합하도록 확장한 것이다. 이 모델을 바탕으로 우리는 TLS-SEED 가 ACCE-SEED 안전하다는 것을 보였다. 마지막으로, 테스트 베드 기반 실험을 통해 우리는 TLS-SEED 가 무시할만한 부하만 일으키기 때문에 실현 가능하다는 것을 증명하였다.

주요어: 전송 계층 보안(TLS), 미들박스, 엣지 컴퓨팅, 신뢰 수행 환경(TEE)
학번: 2015-21259

Acknowledgements

먼저 이 논문이 나오기까지 옆에서 가장 고생한 제 아내와 아들에게 무한한 사랑과 감사의 마음을 전합니다. 제가 새벽이나 밤늦게 나가서 여러 시간을 연구에 할애하고 매번 학회 논문 기한에 맞춰 살다보니 가까이에 있는 가족들이 많이 힘들 수 밖에 없었습니다. 이 논문이 나올 수 있었던 건 그럼에도불구하고 전적으로 지원해준 가족의 격려가 가장 컸습니다. 또한 제게 늘 사랑을 주시고 지원해주신 어머니와 집안의 버팀목이신 제 외할아버지께도 감사의 인사를 드립니다.

네트워크 보안 해보겠다고 무작정 연구실에 인턴으로 들어온 지 어언 햇수로 6년이 흘렀습니다. 제 지도교수님이신 권태경 교수님은 제가 아무런 배경이 없이 왔음에도 저를 받아주시고 박사까지 키워주셨습니다. 특별히 감사하다는 말씀 드립니다. 최양희 교수님 또한 연구실에서 많은 조언을 주셨고 제 논문 심사의 심사위원장을 기꺼이 맡아주셨습니다. 특히나 좋은 논문이 되었을 때 격려의 말씀도 주셨는데 매우 감사드립니다. 또한 제 박사 논문 심사에 심사위원으로 참여해주신 백윤흥 교수님, 이병영 교수님, 그리고 고려대 허준범 교수님 모두 진심으로 감사드립니다. 이 분들의 조언으로 제 학위논문이 보다 발전하여 세상에 나올 수 있었습니다.

먼저 제가 쓴 논문과 관련하여 직접적으로 많은 도움을 준 선후배님들에게 진심어린 감사의 마음을 전합니다.

세린이와 (임)정환, 준혁이는 제가 쓴 논문과 관련해서, 그리고 제 개인적인 삶과 관련해서 여러 가지로 많은 도움을 주었습니다. 매우 감사합니다. 준희 형 또한 저와 함께 한 논문 연구에서 많은 기여를 해주셨습니다. 지금 같이 하고 있는 연구도 잘 되리라 믿습니다. 용배 또한 제 논문에 대해 많은 토론을 해주었습니다. 연구실에서 잠시 인턴을 하면서

maTLS 논문에서 실험을 진행한 경재와 TLS-Seed 논문에서 ACCE-Seed 모델 관련 부분을 함께 토론해준 용운에게도 감사의 마음을 전합니다. 위 후배들은 연구실 생활 이상으로 오랜 기간 함께한 인연이지만 특별히 제가 쓴 논문들에 물심양면으로 많은 기여를 해주었습니다. 진심으로 감사합니다.

또한, 제 연구 분야의 개척자라고 할 수 있는 CMU/Nefeli Networks의 David Naylor는 저에게 많은 조언을 아끼지 않았습니다. 그리고 우연히 저희 연구실에 와서 알게된 Zach Smith는 저에게 Tamarin을 알려주었으며 여러 연구도 함께 하게 되었습니다. 연구실 선배로서 현재 미국에서 교수를 하고 있는 태중이도 출중한 글 솜씨로 maTLS 논문이 질적으로 발전할 수 있도록 해주었습니다. CCS에서 알게된 인연이 지금까지 이어져 공동 연구를 하고 있는 두원이 형도 여러 조언을 주었습니다. 모두에게 감사하다는 말씀 드립니다.

연구실의 보안팀에게도 감사의 마음을 전합니다. 우리 연구실에서 보안 연구를 수행한 역사가 길지 않지만 양적, 질적으로 성장할 수 있었던 것은 보안팀이 보안 세미나를 비롯하여 여러 프로젝트로 공동 연구를 수행했기 때문일 것입니다. 현재 연구실에서 계속 연구를 수행하고 있는 맏형이신 은상이 형과 민경이, 현민이와 민혁이를 비롯하여 같이 공부와 연구를 진행한 졸업생 여러분들께도 감사하다는 말씀 드리고 싶습니다.

그리고 박사 심사 과정에서 많은 도움을 준 명철이와 컴퓨터연구소에서 같은 공간에서 연구를 함께하면서 공동 생활을 해온 (송)정환이와 동현이, 측위 연구를 이어가고 계시는 초롬 누나, 군에서 연구를 위해 우리 연구실에 온 윤교, 그리고 같이 프로젝트하면서 최선을 다하고 있는 상윤이 모두 감사드립니다.

이상 언급한 많은 분들 외에도 제 박사 과정 동안 많은 조언을 주시고 격려를 아끼지 않으신 많은 교수님들과 연구실 선후배님들, 그리고 여러 인연들이 있습니다. 모두에게 감사의 마음을 전하며 하시는 일들 모두 잘 되시기를 기원합니다.