TOWARDS USABLE AND FINE-GRAINED SECURITY FOR HTTPS WITH MIDDLEBOXES

Abhimanyu Khanna

A thesis

in

The Department

of

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science

in Information Systems Security at

Concordia University

Montréal, Québec, Canada

May 2017

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Abhimanyu Khanna**

Entitled: **Towards usable and fine-grained security for HTTPS with middleboxes**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Jia Yuan Yu _____ Chair

Dr. Walter Lucia _____ Examiner

Dr. Emad Shihab _____ External Examiner

Dr. Mohammad Mannan and Dr. Jeremy Clark _____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 2017 _____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Towards usable and fine-grained security for HTTPS with middleboxes

Abhimanyu Khanna

Over the past few years, technology firms have inlined end-to-end encryption for their services and implored for increased in-network functionality. Most firms deploy TLS and middleboxes by performing man-in-the-middle (MITM) of network sessions. In practice, there are no official guidelines for performing MITM and often several tweaks are used resulting in less secure systems. TLS was designed for exactly two parties and introducing a third party by doing MITM breaks TLS and the security benefits it offers.

With increasing debate in finding a clean way to deploy middleboxes with TLS, our work surveys the literature and introduces a benchmark based on the Usability-Deployability-Security (UDS) framework for evaluating existing TLS middlebox interception proposals. Our benchmark encompasses and helps understand the current benefits, solutions and challenges in the existing solutions for incorporating TLS with middleboxes. We perform a comparative and qualitative evaluation for the schemes and summarize the results in a single table. We propose: Triraksha, an alternative to the currently deployed middlebox interception models. Triraksha provides a packet inspection service for end-to-end encrypted connections while maintaining fine-grained confidentiality for end points. We evaluate a prototype implementation of our scheme on local and remote servers and show that the overhead in terms of

latency and throughput is minimal. Our scheme is easily deployable as only a few software additions are made at the middlebox and client end.

Acknowledgments

At the end of this thesis, I would like to thank all the people who helped me during the course of my Masters program.

First and foremost, I would like to thank my advisors Dr. Mohammad Mannan and Dr. Jeremy Clark. You have been tremendous mentors and role models to me, always guiding me on the right path and supporting me when needed. I appreciate all your contributions of time, ideas, and funding to make my Masters experience productive and stimulating. Your mixture of advice, inspiration and criticism on both research as well as on my life career have been invaluable in helping me realize my strengths and weaknesses. My learning curve has only ever steepened under your guidance. You created infinite space and freedom for me to find my own path in life and research and for this I shall be ever grateful.

I believe that I am extremely fortunate to have worked with two outstanding individuals and scholars - Lianying Zhao and Xavier de Carné de Carnavalet. With your confidence, focus, kindness and friendship, you embed in me qualities of sincerity and dedication. My interaction with you has developed in me clarity of thought and systematic approach to problems in research. Thank you for all your feedback and encouragement. I would also like to thank all fellow lab mates and the other faculty members of the CIISE department. I enjoyed a lot from our discussions and learned a lot from the courses that I attended during my Master's program.

Finally, I would like to dedicate this thesis to my family. Words cannot express how grateful I am for their support throughout the course of my Masters degree. I thank my parents Sanjay Khanna and Jasmeet Khanna for educating me with their ideals, for unconditional love and encouragement to pursue my interests, even when the interests went beyond boundaries of my field. I would not have been able to reach this position in my life without them. My parents have truly shown me the essence of never giving up and how we never fight our battles alone. I thank you for your help to handle all the difficulties and wrong decisions I made. I have experienced your guidance day by day and I will keep on trusting you for my future. I would like to thank my brother - Abhinav Khanna for instilling ambition, cheering me and motivating me throughout the course of my Masters.

Abhimanyu Khanna

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

With advances in Internet technology, there is increased usage of middleboxes in networks [59, 80, 24]. Functionalities offered by middleboxes like firewalls, NATs, proxies provide a wide range of services benefiting end users and network operators. Services like caching, compression, prefetching and load balancing improves page load time, data usage and reduces consumption of resources [39, 82, 43, 59] on end points. Network Intrusion Detection systems (IDS)/Intrusion Prevention systems (IPS) and network scanners like Snort [16], Suricata [17] and Bro [63] prevent network attacks and help detect malware/viruses in packet payloads. Parental filtering devices and policy enforcement appliances [13, 74] help mandate and enforce policies on browsing of network traffic and data usage. Middleboxes have become an essential component [83, 50, 23] in the network of many organizations and enterprises. However, trends [72, 70, 69, 62, 6] also show that usage of middleboxes is associated with high cost, complex management and a host of privacy concerns.

Recent data breaches and increasing concerns for user's privacy [48, 84, 67, 79, 2, 33, 77] have led to increased use of end-to-end encryption. This trend has led to the widespread adoption of web servers using HTTPS as the norm to communicate with clients. It was forecasted that 70% of global Internet traffic will be encrypted in 2016, with many networks exceeding 80% [22, 60, 58]. TLS (Transport Layer Security) has become an intrinsic component of HTTPS services and provides for a secure communication channel between a client and server. TLS was designed for two parties and ensures the following properties for end points: *entity authentication*, *data integrity* and *data secrecy*. Middleboxes, which perform in-network functionality at various points in the network [58, 44] do not work well with end-to-end encrypted sessions. Use of TLS pushes any in-network functionality employed by a network operator to be done at the application layer of the end points. Services like packet inspection and payload manipulation cannot be performed as the middlebox does not have access to the packet payload. The very benefits offered by HTTPS are in variance and block the essential in-network functionality of middleboxes.

To get around end-to-end encryption and enable middlebox functionality, middlebox systems are deployed in an insecure method called 'Split TLS' [59]. The middlebox is placed as a gateway in the network and simply performs MITM (man in the middle) for all TLS connections. The middlebox pipes data between the client and the server in two separate TLS connections. Split TLS though widely deployed, has several downsides emanating from its design. A major downside is that the client has to trust the middlebox to securely connect to the server on its behalf. Further, the middlebox has complete access over the data that was meant only for the end points and can read/modify it. Split TLS violates the end-to-end security guarantees of TLS. This is a cause for concern considering the recent number of data breaches by hackers. A number of issues were recorded and surveyed in [58, 44] expressing

privacy concerns for users.

Using end-to-end encryption with in-network functionality of middleboxes is a recent topic of interest and has caused for increasing debate [19] in the community. Many RFCs and academic proposals [59, 47, 73, 72, 64, 55, 53] attempt to construct a protocol that allows middleboxes to exist side-by-side with end-to-end encryption. Recent proposals like mcTLS [59], Blindbox[73] and Embark [49] discuss, raise issues of data permissions and privileges associated with a middlebox in a TLS connection and attempt to address these issues in their scheme. The schemes propose to extend TLS, outsource in-network processing to cloud services and/or propose new searchable encryption schemes. The privacy model in these schemes is stricter when compared to Split TLS. Industrial efforts by Akamai [42, 47], Google [64], Ericsson and AT&T [53] also attempt to bring forward solutions to incorporate middlebox infrastructure with end-to-end encryption. However, the solutions in the existing literature still suffers from limitations. A majority of the schemes are not compatible with the server and are hence less likely to be adopted. Schemes have varying assumptions on privileges for the middlebox and usability for end users. The approach taken by a few schemes incur overhead [75, 52] that may not be suitable for all environments like mobile networks and in networks of content service providers. Further, many of the proposed schemes are not evaluated extensively over real world data points. A good solution to the problem is yet to exist and the topic is very active with various discussions in the IETF, TLS and middlebox community.

1.2 Contributions

In this work, we begin with a survey of the literature with the aim of understanding:

1. The problems and challenges of deploying middleboxes with end-to-end encrypted sessions in current proposals.

2. The properties required for a middlebox to work securely with end-to-end encryption.

We propose a benchmark for comparative evaluation of the existing literature. The benchmark defines 12 properties fitting to the UDS (Usability-Deployability-Security) framework and is used to evaluate 12 proposals from the academia and industry. The results were summarized and placed in a comparative table. We further discuss and demonstrate with each property how and why the schemes are rated as such. The systematical exercise done in our study establishes and provides for insight into the challenges, problems and solutions for use of middleboxes with end-to-end encrypted sessions. Based on a comprehensive survey and understanding of using middleboxes with TLS, we present Triraksha: an alternative scheme to Split TLS. Triraksha uses existing infrastructure to securely incorporate a packet inspection service for TLS connections in enterprise environments. Our scheme Triraksha achieves the following:

1. It requires minimal changes at end points and is deployable with existing infrastructure.

2. Introduces fine-grained security for a client taking part in a TLS connection with a server.

3. An in-network packet inspection service.

4. Incorporates the security properties of TLS and has a stronger privacy model compared to Split TLS.

We implemented Triraksha by adding software modifications only on the middlebox and the client. Our evaluation shows that Triraksha incurs little overhead when compared with a regular end-to-end TLS connection.

In summary, our contributions are as follows:

1. A benchmark based on the UDS framework to assess the benefits of a scheme that attempts to incorporate middleboxes in end-to-end encrypted sessions.

2. A comprehensive study of the existing literature and a comparative evaluation of the schemes using our benchmark.

3. Triraksha: a practical alternative scheme to Split TLS for inspection of TLS traffic in enterprise environments.

4. A prototype implementation of Triraksha in a controlled environment.

5. Strategies and discussion for extending Triraksha to future versions of the TLS protocol and working under extended threat models.

1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 covers some necessary background and literature related to this dissertation. In Chapter 3, we introduce our benchmark based on the UDS framework; summarize the schemes and do an evaluation of schemes with our benchmark. Chapter 4 discusses the threat model, design, implementation details and evaluation of our scheme Triraksha. In Chapter 5, we discuss extensions and general concerns for Triraksha. Chapter 6 concludes.

Chapter 2

Background

In this chapter, we discuss the TLS protocol, introduce middleboxes[2], discuss a few existing classification techniques for middleboxes and summarize deployment of middleboxes in networks using Split TLS. The section establishes that middleboxes are beneficial and provides an overview on why it is necessary to work on a solution to cleanly deploy middleboxes with a higher layer protocol like TLS.

2.1 Transport Layer Security (TLS) protocol

The TLS protocol was designed to provide secure communication for a connection between a client and a server. TLS provides confidentiality and integrity of messages for higher layer protocols like HTTP, IMAP, SMTP etc. The terms TLS and SSL are often used interchangeably but the protocols differ subtly. TLS 1.0 was the successor to SSL 3.0 and the current version of TLS is 1.2 [37] which is also the most widely deployed TLS version. Version 1.3 [38] is currently under development.

[2]Formal definition of the terms used throughout the paper are provided in the Appendix.

TLS is placed at layer 5 in the TCP/IP [12] network stack and consists of the handshake protocol and the record protocol. The handshake protocol is responsible for setting up a secure session and establishing symmetric encryption keys between the server and the client while the record protocol uses the symmetric encryption key from the handshake protocol to do encryption and decryption of TLS record packets. TLS packets are of four content types: handshake, cipher key exchange, application data and alert. Each content type has its own header values and are summarized in the Appendix. The TLS protocol is build on top of various cryptographic primitives and tasks like key agreement, authentication, encryption and integrity protection. The protocol allows the end points to choose from a list of algorithms to perform these tasks. The collective selection of algorithms are referred to as a cipher suite. An example of a cipher suite is TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 where ECDHE_RSA is the key exchange algorithm, AES_128_GCM is the bulk encryption algorithm and SHA256 is the message authentication code algorithm.

TLS ensures the following properties:

1. *Entity authentication*: An end point authenticates who it is communicating with.

2. *Payload confidentiality*: which allows only the two parties in the connection access to the data sent between them.

3. *Payload integrity*: which provides for data integrity and authenticity of the messages communicated (in transit) between the parties.

We summarize the handshake and the record protocol in the following subsection.

2.1.1 The handshake protocol

The TLS handshake is responsible for establishing TLS session keys and authenticating two end points in a connection. Figure 1 represents the TLS handshake done using RSA. A TLS handshake is initiated by a TLS client with a ClientHello message. The ClientHello contains information about the TLS version and a list of supported cipher suites. The server responds with a ServerHello message containing the selected cipher suite. The server also sends the server Certificate, ServerKeyExchange and a ServerHelloDone message. The ServerHelloDone message indicates the end of the transmission of the ServerHello. The client in turn sends a ClientKeyExchange, ChangeCipherSpec and Finished message.

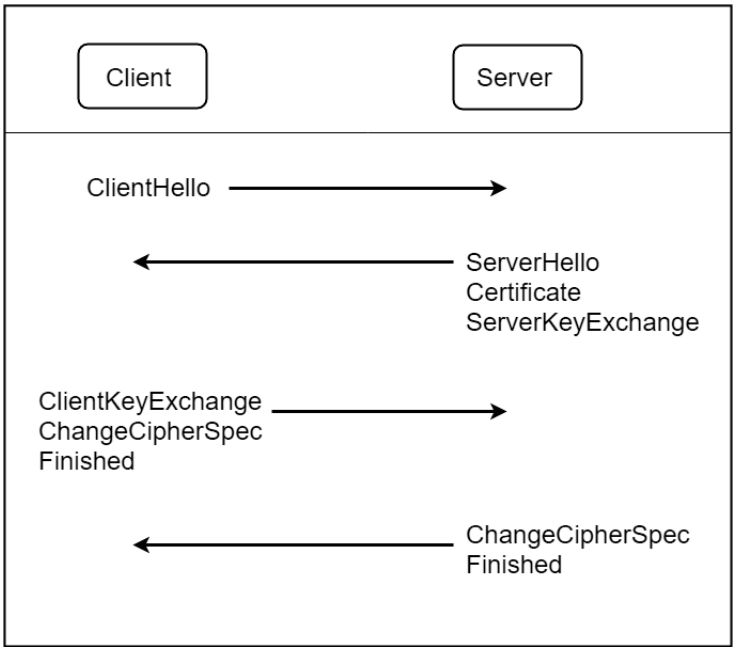

Figure 1: Example of a TLS handshake done using RSA.

The information in the Client/ServerKeyExchange messages are used to determine the TLS session keys (TLS sessions keys are used for encryption of TLS application data). The ChangeCipherSpec message indicates that all subsequent messages sent between the endpoints would be encrypted. Following the Client Finished message,

the server responds with the ServerChangeCipherSpec and Server Finished message. The Finish message is a hash of the entire handshake and ensures that the previous messages have not been tampered with. A variation to the TLS handshake involves session resumption where the end points can resume a previously negotiated session.

2.1.2 The record protocol

The record protocol is used to transmit protocol messages. A TLS client takes messages from a higher layer and uses the cipher suite negotiated during the handshake to encrypt the message. The TLS client based on the cipher suite usually computes a MAC over the plaintext, pads the message to a fixed block length and finally encrypts it to get ciphertext. The ciphertext is wrapped with the protocol version, message type and message length headers and passed to a lower network stack.

2.2 Middleboxes

2.2.1 Introduction

Middleboxes are units that stay in the network between two end points and perform some functionality on the traffic by altering, inspecting, filtering or transforming it. Significant changes have been made to the technology used in the Internet in the last few years and with higher processing power of computers, performance and security expectations from commercial services, content service providers and regular servers have increased [36, 25]. Middleboxes provide the necessary functionality [83, 50] to meet them. Compared to the original architecture of the Internet [32] which is based on the end-to-end principle and pushes functionality at end points,

middleboxes provide functionality and increase resource efficiency in-network. Middleboxes helps the Internet grow into a market driven ecosystem [59]. Middleboxes are widely deployed in networks of all sizes. All major Internet service providers, content service providers, mobile carriers, enterprises use middlebox for a wide range of services. Not all functionality transpired by middleboxes is desirable as there are increasing privacy concerns [77, 33, 2] on how middleboxes handle traffic. However, it is established that middleboxes are essential and provide the necessary resources for end points to perform better on the stage [49, 82].

2.2.2 Middlebox taxonomy

RFC 3234 [32] introduces a taxonomy for middleboxes. The most common way to classify a middlebox is a classic: "good versus bad" scenario. Some middleboxes provide useful functionality while some are malicious and attempt to disrupt communication between the endpoints. Middleboxes are also classified at the protocol layer depending on which layer they provide functionality in. mcTLS [59] depicts examples of application layer middleboxes and the permissions it needs for functionality at the HTTP layer. The examples are a good representative of the most common types of middleboxes in networks. They are: cache, compression, load balancing, IDS, parental filter, packet pacer and WAN optimizer. They classify middleboxes into three categories: value added services (users opt in for these middleboxes), administrator mandated (helps enterprises/institutions to set policies) and unauthorized (not useful for network or user). TLS-AUX [47] classifies middleboxes into content aware and non content aware services. Content aware services deal with the packet payload while non content aware services usually work on the headers and other meta data associated with the connection. Middleboxes under the content aware services category can access the plaintext data sent between the client and the server. Some examples are

an Intrusion detection systems, Intrusion prevention systems and content filters. In our scheme Triraksha, we are interested in application layer middleboxes that provide useful functionality such that the middlebox is capable of monitoring and inspecting TLS traffic but not modifying it.

Middlebox operators and environment: We briefly summarize the environment and the operators that run the middlebox services. In medium and small sized networks like home/company networks, middleboxes which typically provide functionality at an application layer are operated as proxies or application layer gateways (ALGs) while middleboxes which provide functionality at IP layers and below commonly run as routers, firewalls and NAT (network address translator) devices. A proxy is a device that relays application messages between two peers in the network. Proxies are capable of terminating sessions with the client and the server, acting as server to the end-host client and as client to the end-host server. Proxies are of two types: forward proxy and reverse proxy. A forward proxy forwards requests from a peer to another in the network while the reverse proxy is used as a front-end to control and protect access to a server on an network. Both forward and reverse proxies are typically used to provide services like anonymity, compression, load balancing, WAN optimization, SSL acceleration etc. Application layer gateways are entities programmed to provide a specific set of services and are implemented as hardware or software at the gateways in an network. ALGs may reside within the gateway device or reside externally but communicating with the gateway through a protocol. Contrary to proxies, ALGs do not terminate connections between its peers. ALGs inspects or optionally modifies application payload content to provide the middlebox service and continue the flow of application traffic as a network hop. In large scale networks, middleboxes are collectively run at multiple points to provide a wide array of functionality. The collective distributed infrastructure for middleboxes services in

large scale networks are typically run as a CDN. A CDN has a large number of surrogate servers in geographically different locations. A website using a CDN service would have its data replicated to the surrogated servers and when users access the web site, they will be directed to the CDN and finally get the content from a nearby surrogate server rather than the web site's origin server. CDNs originally were designed to reduce network latency but have evolved to provide services like DDoS protections appliances and application layer firewalls.

2.2.3 Split TLS and why it breaks regular TLS

A naive and popular solution used to incorporate middlebox environments with TLS is to place the middlebox transparently between the end points. The middlebox acts as a certificate authority and provides a self signed root certificate for each client in the network. The client installs this root certificate on his end and when the client attempts to connect to a server, the middlebox intercepts the connection and fabricates a brand new certificate for that server (the fabricated certificate is signed by the key of the root certificate installed on the client computer). Since the fabricated certificate is signed by a key that the client trusts, the client application will accept the connection. A second TLS connection is opened with the server and the middlebox then pipes the data between these two end points.

Problems stemming from Split TLS: Split TLS only works for middlebox operators that explicitly install a root certificate on the end point (typically client side). Split TLS is designed insecurely and breaks the security properties offered by TLS in the following ways:

1. *Server authentication:* The client as an end point does not authenticate the

server. To a regular user, no certificate warning signs would be displayed for invalid server certificates on a browser because the browser accepts the certificate signed by one in its trusted root directory. Further, the server is completely unaware of the middlebox in the connection.

2. *TLS negotiations:* The client does not negotiate the TLS handshake parameters with the server and has no security guarantees of the parameters negotiated between the middlebox and the server.

3. *Over privileged middlebox:* The middlebox gains access and is capable of reading and modify the data sent between the client and the server. Full disclosure of user and server traffic increase risk of privacy leak in systems.

Adopting Split TLS leads the user to have a false sense of belief that the client is communicating with the server when in-fact the client is transparently communicating with the middlebox. The bottom line is that Split TLS brings about risk and results in less secure systems. Despite its limitations this scheme is widely used because of its ease of deployment.

In the subsequent chapters of this thesis, we propose our scheme: Triraksha, which is an alternative to Split TLS and provides better security benefits. We briefly discuss how Triraksha has better security benefits over Split TLS. The Triraksha protocol provides the following benifits:

1. *Server authentication:* The client and the server perform a TLS handshake. The middlebox does not participate in the TLS handshake. The client sees the certificate of the server and trusts this certificate if it is signed by a root certificate on the client computer.

2. *TLS negotiations:* Compared to Split TLS, where the middlebox negotiates the TLS version and the ciphers for the connection with the server (the middlebox may degrade the security of the TLS connection by using poor or broken ciphers). In Triraksha, the client sets the list of ciphers and TLS version to be used for the connection.

3. *Over privileged middlebox:* Compared to Split TLS where the middlebox has access to the whole traffic (the middlebox can modify traffic or do unwarranted actions like insert ads etc.), In Triraksha, the client controls for which connections the middlebox can read the traffic. The middlebox cannot modify the traffic in Triraksha and maintains content integrity for the messages sent between the client and the server.

2.2.4 The world of middleboxes with end-to-end encryption

In the previous section, we discussed that middleboxes are used with TLS using Split TLS. However, this works for only middleboxes that have an close association with the end points (as the end point installs a certificate from the middlebox). Middleboxes that operate without such a relationship must reveal their presence explicitly. It is possible for middleboxes to coexist with end-to-end encryption without solely relying on the Split TLS model but they must get consent/be trusted by atleast an end point of the connection (client or server). Trust and explicit presence of the middlebox are two necessary properties required for a middlebox to coexist with end-to-end encryption. A middlebox that is not visible to an end point cannot be trusted by that end point [40]. The world of middleboxes existing with end-to-end encrypted connections are described in Figure 2.

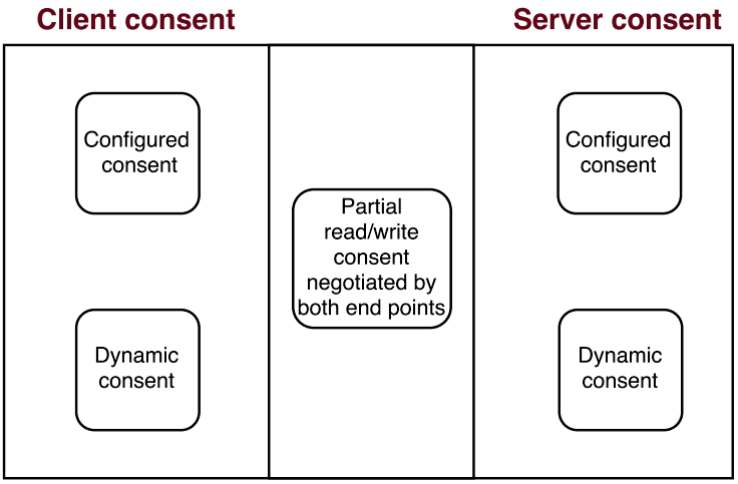

Figure 2: Classification of schemes for middleboxes coexisting with TLS.

The world described in Figure 2 contains the client (an end point that wants service), the middlebox (which provides network functionality) and the server (the service provider to the client). The environment is roughly divided into either the client or the server providing consent to allow the middlebox in the connection. Configured client side [40] consents to grant the middlebox access to traffic in an end-to-end encrypted connection by allowing the middlebox to MITM the connection. A typical example of such a model is Split TLS. The middlebox has access to all traffic and is transparent to to the server. A variation to the 'Configured client side model' is the 'Dynamic client side model' where the client learns the presence of the middlebox after the clients attempts to contact the service in the respective network. In this case, the client receives a certificate for the middlebox and the client can proceed with the connection or not explicitly granting consent to the proxy. Contrary to the client providing consent, there are models which allow the server to provide consent in the similar manner. The client is transparent to the middlebox in this case. An example of such an model is Keyless SSL [34]. We describe these models in the subsequent sections. Recent proposals and models involve the client and the server to negotiate and allow a predefined list of middleboxes or allow for middlebox discovery.

Following the negotiation, the client and server grants read or write permissions to the middlebox on components of the traffic. An examples of such a model is mcTLS [59].

Chapter 3

UDS framework

In this chapter, we propose a benchmark to assess schemes which incorporate middleboxes in TLS connections. We follow up with a comparative evaluation of the schemes using the benchmark and discuss how and why the properties are rated based on facts or any assumptions made in the proposals.

The UDS framework was introduced in [31] and provides a semi-structured way to comprehend the benefits of a scheme. The properties chosen in the framework allows the schemes to be [31] "*rated across a common, broad spectrum of criteria chosen objectively for relevance in wide ranging scenarios, without hidden agenda*". The entities and context for trust in various schemes differ widely in assumptions made in their privacy model, user consent and deployability for operators. It is important to understand the process and the end goal a scheme tries to achieve. By introducing a benchmark that encompasses the properties of the schemes, we can align the security goals with the deployment needs of operators and usability for end users. The benchmark we propose addresses the assumptions made in the literature, the trust between the entities, their requirements and what they accomplish.

Schemes can be broadly classified into categories based on which end point would provide consent to include the middlebox in the connection. Consent can be provided by either the client or the server. In some schemes, it is assumed that the end user is aware of the middlebox in the network by having an explicit list of the middleboxes that they would allow to participate in the connection (the list in few cases is agreed during handshake by the server and the client) while, with some schemes the end users make policy decisions on discovery of a middlebox. The policy might apply to each connection or for a longer period; granting specific permissions or complete read and write permissions on the data.

Our classification is based on which end point trusts the middlebox and the communication strategy used between the end point and the middlebox. We categorize schemes into two categories (scheme either provide client side consent or server side consent to the middlebox) and the client side consent schemes are further classified into three categories. These categories are:

1. Proxy based schemes: In this category, the middlebox acts as a proxy and terminates the connection between the end points. The consent to add a middlebox can be provided by the client or the server and the proxy pipes data between the end points.

2. Passthrough: In this category, the middlebox allows the end points to communicate directly with each other. An endpoint then uses a custom protocol to communicate with the middlebox and provide consent. The communication protocol with the middlebox maybe inband or out-of-band with the connection to the other end point.

3. Modified TLS: In this category, TLS is modified to do a three party key exchange during the handshake. The end points agree on a predefined list of middleboxes

to use in the session or use an extension with the TLS ClientHello/ServerHello for middlebox discovery. An end point can provide consent to the middlebox to read/write components of the traffic after the initial negotiation.

3.1 UDS benchmark

The benchmarks are divided into three categories: Usability, Deployability and Security. For the rest of the paper, a property shall be addressed to with an italicized mnemonic title.

The schemes are rated qualitatively as either supporting the property or not supporting the property. In some scenarios, sometimes the property is not applicable and is hence rated accordingly. Cases in which it is unclear if the property is supported by the scheme or not, we make an assumption fitting to the scheme or give it the benefit of the doubt and rate it to have a partial compliance for the property. Such schemes generally lack sufficient technical details in their proposal. We discuss when such an assumption is made for the scheme. The decision to evaluate the scheme qualitatively rather than quantitatively (assigning a weight to each scheme and ranking them linearly) is motivated by the fact that we would like to have a understanding of the technical issues for using middleboxes in TLS and focus on solving those problems rather than ranking schemes linearly. The ratings illustrate only what a scheme achieves. If a scheme supports a particular property, it does not mean that we endorse the scheme to perform better than its alternatives. We now proceed to describe each of the properties.

Usability

1. ***Middlebox discovery visibility***: The client/user is aware of the existence of

a middlebox when they make a connection to the server.

This property is not applicable for schemes that provide server side consent.

Rating for UDS framework:

Yes: The scheme shows the existence of a middlebox to the user/client during each TLS connection.

Partial compliance: The scheme shows the existence of a middlebox is only shown to the user/client initially when the middlebox is configured.

No: The scheme shows the existence of the middlebox is not shown to the user/client during a TLS connection.

2. ***Middlebox persistence visibility***: This property checks if the middlebox is visible to the client/user when the user switches to a new network.

   This property is not applicable for schemes that provide server side consent.

   Rating for UDS framework:

   Yes: The middlebox defaults to intercept each connection and show its existence to the user/client when the user switches to a new network.

   No: The middlebox defaults to not intercept a connection when the user switches to a new network.

3. ***Infrequent errors or open fails***: The property puts in check if the scheme is capable of handling open fails and supports error handling when the end parties rely on alternative authentication methods (key pinning, DANE, HSTS etc.) other than certificate validation.

   Rating for UDS framework:

   Yes: The scheme has a feature that allows the client to handle open fails. The

client, middlebox and server should be capable of supporting HTTPS authentication services other than certificate authentication.

No: The scheme has no feature to handle open fails.

Security

1. ***Server authentication***: The client authenticates the server in the handshake phase. Server authentication should be achieved before any TLS application data is transmitted between the client and the server. The scheme should always let the client authenticate the server's certificate.

   Rating for UDS framework:

   Yes: The scheme allows the client to authenticate the server.

   No: The client does not authenticate the server.

2. ***Middlebox recognition***: Any middlebox that would intercept the connection between a client and a server should be recognized by the client.

   Rating for UDS framework:

   Yes: The client recognizes the middlebox.

   No: If the client does not recognize the middlebox.

3. ***Connection specific interception***: The scheme allows fine-grained data confidentiality for each connection between the end points and the middlebox i.e. trusting a middlebox for one connection should not extend the trust to subsequent and future connections. The end points can make several connections and the middlebox (or a number of middleboxes in the network) may or may not be able to perform functionality for all of these connections (based on the policy set by the client/server).

   Rating for UDS framework:

Yes: The scheme has a mechanism to set fine-grained access control on each connection made by the end points.

No: The scheme has no mechanism to set fine-grained access control on each connection made by the end points.

4. ***Minimal read disclosure***: The scheme supports least privilege for read access on data in a connection which is intercepted by a middlebox. The middlebox should be provided with the least read level access it requires to perform its functionality. Read access can be classified into three levels: full read access, partial read access and no read access. A scheme allows full read access if the middlebox can read and inspect all data between the client and the server. A scheme is marked as partial read access if a middlebox is allowed to read only selective data sent between the client and the server, for example, in a partial read access scheme, a middlebox would support keyword inspection giving the middlebox privilege to read only few keywords. Other examples would include schemes in which a middlebox has privileges set by the end points for the data it can read. A scheme is marked no read access if the middlebox cannot read the data communicated between the end points.

   Rating for UDS framework:

   Yes: The scheme provides no read access to the middlebox for the data transmitted between the client and the server.

   Partial compliance: The scheme provides partial read access to the middlebox for the data transmitted between the client and the server.

   No: The scheme provides full read access to the middlebox for the data transmitted between the client and the server.

5. ***Minimal write access***: A scheme should have minimal write access to the data accessed by a middlebox. A middlebox should be provided with write

access to the data only if required. The scheme at any given phase should be able to maintain message integrity between the end points. The client and server should be able to detect modification made to the traffic between them. Write access can be classified into three levels: full write access, partial write access and no write access. Full write access implies that the middlebox can modify data before it pipes it between the client and the server. Partial write access means the scheme supports selective modification to data that is sent between the client and the server. No write access means the middlebox does not have access to the data communicated between the client and the server.

Rating for UDS framework:

Yes: The scheme provides no write access to the middlebox for the data transmitted between the client and the server.

No: The scheme provides full write access to the middlebox for the data transmitted between the client and the server.

Partial compliance: The scheme provides partial write access to the middlebox for the data transmitted between the client and the server.

6. ***Client negotiation for TLS***: This property puts in check if the client is allowed to handle TLS handshake negotiations with the server.

   Rating for UDS framework:

   Yes: The scheme allows the client to negotiate all handshake parameters for a middlebox in the connection.

   No: The scheme does not allow the client to negotiate all handshake parameters for a middlebox in the connection.

   This property is not applicable for schemes that provide server side consent.

Deployability

1. ***No significant latency***: The property checks if the scheme has additional overhead than a regular TLS connection. Overhead can be measured in term of extra round trips, page load times, handshake sizes (and file download times for varying configurations of link speed and file size). In our evaluation, we compare the number of additional TLS handshakes between the entities in a scheme with the entities (client and server) in a regular end-to-end TLS connection. A TLS handshake in the scheme maybe extended (requiring additional information to be carried between the entities or be dependent on some process). For this property, an extended handshake is the same as a regular end-to-end TLS handshake and we ignore the additional network latency of the extended TLS handshake.

   Rating for UDS framework:

   Yes: A scheme is rated yes if it has equal number of TLS handshakes than that of a regular end-to-end TLS connection.

   No: A scheme is rated no if it has more number of TLS handshakes than that of a regular end-to-end TLS connection.

2. ***Server compatibility***: This property keeps in check if in a scheme any changes are made at the server end or additional extensions are needed to the TLS protocol itself. An user (organization/enterprise) at the server end would less likely adopt another scheme if additional modifications are required at their end.

   Rating for UDS framework:

   Yes: The scheme requires no changes to be made at the server end.

   No: The scheme requires changes to be made at the server end.

3. ***Middlebox compatibility***: This property checks if the scheme requires middlebox providers to make changes at the middlebox end. The scheme would be easier to use if it supports extensions/plugins to middleboxes and can be deployed with existing infrastructure. If a non mature scheme were to be adopted, in-network processing would require middlebox providers to update their software/hardware.

   Rating for UDS framework:

   Yes: If no changes are required at the middlebox end.

   No: If changes need to be made to the middlebox for it to be usable in the scheme.

3.2 Summary of schemes evaluated using UDS framework

We use the benchmark described in the previous section to do a qualitative evaluation of TLS middlebox interception schemes. The evaluation process is summarized in Table 1. In this section, we summarize the different schemes according to the category they belong to. We start with schemes that are based on giving client side consent to the middlebox.

3.2.1 Modified TLS schemes

1. **mcTLS:** mcTLS [59] proposes to let the client and the server agree on a middlebox/list of middleboxes that should be authorized in the connection to read or write certain parts of the traffic. They introduce the notion of contexts. An context is a set of symmetric encryption and/or message authentication code (MAC) keys for controlling who can read and write the data. In other words,

the context is a set of privileges and these privileges allow the data to be read or modified by the middlebox. Applications can associate each context with a purpose and access permissions for each middlebox. For instance, web browsers/servers could use one context for HTTP headers and another for content. The client and the server decide for each context its privileges and accordingly the middlebox gets access to keys and can read (encrypt/decrypt) or write (encrypt/decrypt and authenticate) over the context. They introduce a three way key-exchange protocol for their scheme by extending the TLS handshake to establish ephemeral session keys for each party taking part in the protocol. The paper discusses the security model for middleboxes in a TLS session, the relevant permissions required by a middlebox in a TLS session and the impact of using contexts on privacy of the TLS session in mcTLS.

2. **Blindbox:** The Blindbox [73] proposal introduces the usage of two party computation to implement secure keyword inspection by middleboxes in an end-to-end TLS connection. The key difference between mcTLS and Blindbox is that in Blindbox, the middlebox can read only keywords (generated by a third party called as rule generator) in the traffic. The middlebox cannot modify the traffic. The scheme use different cryptographic protocols to achieve their goals altogether.

Keyword inspection allows the middlebox to detect whether the encrypted traffic between the client and the server matches a particular pattern or keyword. In the proposal, two streams of TLS connections are set up. The first stream allows the client to communicate with the server directly. For the second stream, all data sent by the client is divided into a number of tokens. These tokens are encrypted by the client and sent to the middlebox. An entity: 'Rule generator'

(trusted by the endpoints), generates a list of keywords (or rules) for which it wants the middlebox to inspect the encrypted traffic. Using oblivious transfer and two party computation, the middlebox is permitted to inspect the traffic for particular keywords without it gaining access to the plain text. The middlebox can only inspect the traffic for particular keywords that match the attack rules established by the rule generator. The rules/keywords are oblivious to the endpoints.

3. **EFGH (End-to-End Fine Grained HTTP):** EFGH [41] is a scheme that extends the TLS protocol to allows middleboxes to be introduced in a TLS connection between a client and a server. EFGH and mcTLS achieve the same end goals. The key difference between mcTLS and EFGH lies in the approach taken for endpoints to make selective content visible to the middlebox. mcTLS uses the notion of contexts while EFGH uses a custom header in the packet frame.

EFGH introduce a three party key exchange protocol, which upon completion produces four types of encryption keys; two of them are used to encrypt data of which one key is shared between the client, proxy and the server and the other key is shared between the client and the server only. The remaining two keys are authentication keys used to authenticate data between the proxy and the server and the proxy and the client. The authors of EFGH modify the TLS framing format by adding a TLS record header to have a custom frame (also referenced as an EFGH frame). The EFGH frame has a header carrying extra information on the type of the frame. Frames are typically divided into three types: handshake, application data and alerts. Application data frames further contain a header, metadata block and data block. The header in the application

data frame indicates whether the frame's data is visible to the middlebox or not. An EFGH protocol also includes a policy which typically encodes fine-grained disclosure rules. The rules in the policy informs the principals about who (an entity in the TLS session) is allowed to see what (protocol elements).

3.2.2 Passthrough

1. Ubicrypt: The Ubicrypt [81] scheme allows end-to-end encryption between a client and a server while allowing a trusted gateway to inspect traffic. Ubicrypt does not need modifications on the server end. Ubicrypt works on the QUIC protocol (QUIC is an application layer protocol that sits on top of UDP and utilizes a cryptographic protocol similar to TLS for authentication and encryption). At the start of the protocol, the trusted gateway allows an Ubicrypt client to send session negotiation packets to the server until the session encryption keys are generated. The Ubicrypt client securely leaks the QUIC session keys for an connection to the gateway in a separate secure channel. The gateway continues to buffer packets sent by the client or the server and on receipt of the keys will allow the packets to pass through. The scheme however suffers from poor real word performance and was not evaluated over realistic data points. Further, they do not have a proof of concept that performs decryption of the packets or a evaluation of the resources and time required to do so.

   Our scheme Triraksha in concept is similar to Ubicrypt however, we incorporate middleboxes for TLS instead of QUIC and enable decryption of TLS packets. Further, we talk in detail about sharing only TLS encryption keys to provide fine-grained trust in our model.

2. **Sharing record protocol keys with a middlebox in TLS (SRPK):** SRPK

[61] uses session key proliferation to achieve its goals of sharing symmetric session keys between a TLS client and a middlebox. The key difference between SRPK and Ubicrypt is that SRPK is server incompatible. SRPK uses a separate content type "KeyshareInfo" added with the ClientHello for middlebox discovery and negotiation. On agreement with the server (for including a middlebox), it sends the symmetric keys. Ubicrypt simply resorts to using a trusted gateway. Further Ubicrypt supports only the QUIC protocol and SRPK supports only TLS.

Specifically, the authors of SRPK construct a 'tls_keyshare extension' that is included in the TLS ClientHello and TLS ServerHello. The extension contains a sequence of SHA-256 hashes of middlebox certificates. The client sends the hashes of the certificates of middleboxes that it wants to include in the session and the server sends a subset of the same hashes for those which it agrees to trust. The RFC defines a new record type 'KeyshareInfo'. This record type allows the client, proxy and the server to agree upon the ciphersuites and TLS version and also contains a data structure to store cryptographic keys. On completion of the protocol, a trusted middlebox will receive the encryption/decryption keys in 'KeyshareInfo' for the data transmitted between the client and the server. They discuss how the client, middlebox and proxy process the 'KeyshareInfo' record at their end.

3.2.3 Proxy based schemes

1. **Split TLS and Split TLS as CA:** We discussed Split TLS in Section 2.2.3. Here we discuss a subtle variation to Split TLS in which the middlebox has as a certificate from a certificate authority or is a certificate authority. Unlike

regular Split TLS where the middlebox simply provides its certificate to the client, here, the middlebox intercepts the connection with a certificate signed by a certificate authority that the client trusts. The middlebox does not have to present the client with its certificate and can intercept the connection by simply being an intermediate in the network.

2. **Explicit Proxies for HTTP/2.0 (Exproxy), R. Peon et al.**: The Exproxy RFC [64] proposed the use of explicit proxies. Their proposal falls under two threat models. In the first threat model, they use a trusted proxy, which is capable of inspecting all data sent and received by the end points while in the second model they use a caching proxy in which only data that can be served from the cache is inspected by the proxy. The middlebox (proxy) intercepts the connection in both Split TLS and Exproxy but the key difference is that Split TLS makes use of certificates to get the client private key while Exproxy explicitly provides the "decryption material" to the middlebox.

   In Exproxy, the client will decide whether it should use a null cipher for encryption of data or give the decryption key for the session to the proxy depending on the security mode. In case of a trusted proxy, the client can use a null cipher for the TLS stream or give the decryption keys for the encrypted data to the proxy while during the use of a caching proxy, the client should not use a null cipher for the TLS stream and not provide the decryption keys for the encrypted data to the proxy.

3. **TLS proxy server extension, Mcgrew et al. (TPS):** The proposal allows proxies to be MITM entities. The key difference between Split TLS and TPS is that TPS is server incompatible and TPS makes use of a custom extension in the TLS handshake to introduce the middlebox in the connection. The custom

extension is also used for giving consent to the middlebox to participate in the connection.

The principal contribution in TPS [55] is to construct a 'ProxyInfoExtension'. According to the scheme, when a client attempts to contact a server, the TLS proxy intercepts and checks if the TLS ClientHello has a ProxyInfoExtension. It then holds the stream data sent by the client. The proxy will then continue to complete a TLS session with the server and send to the client, assertion about the server and the session. The ProxyInfoExtension carries this assertion. The client then performs authentication and authorization processes (checking server certificate, hostname etc.) for the server certificates. The client also authenticates the proxy and establishes trust. On successful run of the protocol, the proxy will be able to relay data between the client and its peer connection. The RFC addresses multiple goals such as handling middlebox discovery, authentication of server and proxy by the client and allowing the client to make access control decisions for the proxy over the content it transmits.

4. **Explicit Trusted Proxy in HTTP/2.0, Loreto et al (ETP):** The RFC [53] describes how an user can provide consent for a trusted Proxy to be securely involved in the connection when the user is requesting an HTTP URI resource over HTTP2 with TLS. Conceptually, there is little difference with ETP and Split TLS. The key difference lies that ETP requires user consent to include the middlebox for each connection.

   In the scheme, when the user has given consent to the presence of the proxy, the client switches to a "proxy mode" in which it does not check the hostname of the origin server against the server's identity as presented in the server certificate

message. The scheme also allows an user to "opt out" and choose to bypass the proxy. Proxy discovery is described in two methods. In the first method, discovery of the proxy is done when the client receives the server certificate (the server certificate contains an 'Extended KeyUsage' extension and a 'proxy authentication key purpose ID'). It performs certificate validation checks and secures consent from the user to allow the proxy in the TLS connection. In the second method, the proxy indicates its presence and identity by intercepting a TLS ClientHello message, and forcing the client to redirect to a secure page on a portal where the user requests to consent to the presence of the secure proxy. In both the methods once the trusted proxy has been identified and user consent is established, the proxy is trusted and has access to all data. Loreto et al. further constrain the trusted proxy such that URIs that are available over the HTTPS scheme do not traverse the proxy. This has the effect of precluding the proxy from performing services that may be of benefit to the user

3.2.4 Schemes that provide server side consent

We first summarize two schemes (configured server side) and then some Industry patents which provide an abstract description on how they use middleboxes in a TLS connection. In this thesis, we do not focus on schemes that use server side consent for middleboxes in detail and do not evaluate them with the UDS benchmark.

1. **Keyless SSL:** Keyless SSL [34] was introduced by CloudFlare and allows for it to have read and write access to traffic between the client and the server. Keyless SSL pushes the consent for access to data to the server side. Keyless SSL lets regular servers retain custody of their private keys while they use CloudFlare to serve traffic. In Keyless SSL, the private key of the server is moved to a

'keyserver'. During a TLS handshake, when the client sends a random pre-master secret encrypted by the server's public key. CloudFlare will forward the encrypted pre-master secret to the keyserver and the keyserver returns an unencrypted pre-master secret (as the keyserver has the private key to decrypt the encrypted random password). Once it receives the random pre-master secret for the TLS session, CloudFlare and the client can derive identical TLS session keys and CloudFlare can serve server traffic encrypted by the TLS session key. The connection between CloudFlare and the keyserver is encrypted by a strong cipher suite.

2. **SSL spliting:** SSL spliting [51] proposed to simulate a SSL connection with a client by combining authentication records from the server with data records from the proxy. The scheme reduces bandwidth load on servers by allowing proxies to serve data which is endorsed by the server. The server signs the data while the proxy is a distribution channel which serves data to the client. SSL splitting does not provide confidentiality, as the proxy has access to the encryption keys shared between the server and the client and uses them to re-encrypt the merged stream. The proposal is hence limited to serving only public data. During the setup phase, the client attempts to initiate an handshake with the server. The proxy will replay the handshake messages to the server and in return the server is authenticated to the client. To respond to the data resource requested by a client, the server sends the MAC of the resource along with a short unique identifier. The proxy splices the payload sent by the server, it uses the unique identifier to lookup the resource in its cache and then merges the MAC from the spliced payload with the resource. It replays the reconstructed stream to the client in a manner indistinguishable to an original stream that would be sent by the server. The server does not have to send any data records

and the proxy has access to the encryption keys but not to the authentication keys. If a proxy does not have a particular resource in its cache then it triggers a cache miss handler which is simple HTTP like protocol that downloads the resource from the server.

3. **Terminating SSL connections without locally-accessible private keys, Akami Technologies:** The patent [42] filed by Akami technologies is very similar to Cloudflare's Keyless SSL. The protocol dictates the SSL handshake and decryption of the pre master secret for the server to be done at a remote location. The server in context with the patent is an edge server in a CDN and not the principal server of the website. The proxy is split to function in a client-server model with the client side residing at the SSL termination point (the edge server). The server component of the proxy resides in a remote place and is associated as a data store in which decryption keys (private keys) are stored. The decryption keys for a server certificate do not reside at the end point (the edge server) and are never accessible from the server component of the proxy. During a SSL handshake, the client proxy component proxies the encrypted pre-master secret it receives from a conventional SSL client and the server component of the proxy returns the decrypted pre master secret. The client component of the proxy forwards this to the edge server with which it can derive master keys and continue with the rest of the handshake normally. The connection between the client proxy component and the server proxy component is mutually authenticated.

4. **Proxy SSL handoff via mid-stream re-negotiation, F5 Networks:** The patent [30] filed by F5 Networks provide an infrastructure in which a single existing SSL connection can be used to serve content from more than one server device. A traffic management device (TMD) is setup between a client and the

first server. To switch to another server the process is as follows: for an existing SSL connection between a client and a server device, the TMD may request a client to re-negotiate an encrypted connection. The criterion for which a re-negotiation request is initiated by a TMD is server configured, for example it maybe a schedule maintenance of a server or based on the type of data requested by a client. The TMD receives the private key from the first server and can decrypt all data sent by the client. The TMD can redirect the responses of the re-negotiation request to a second server device based on network topology, network traffic, server capacity etc. The TMD receives the private key from the second server device, decrypts the responses from the client and may itself becomes the endpoint by encrypting the messages or may forward the responses to the second server device enabling it to be the endpoint.

5. **Accessing SSL connection data by a third-party, F5 Networks:** The patent [66] discusses a method for a proxy intercepting SSL connections which is situated between a client and a server or a client and a traffic management device (TMD). It specifies various embodiments for the connection setup. The secret data to decrypt content in a SSL session can concurrently be given to the proxy or can be given in a separate out-of-band connection. The secret data can be encrypted by the sender and may consist of the pre master secret and server/client random or the master secret. The proxy has access to the payload sent by either end point and can decrypt the data and modify it. The payload may be scanned, logged, audited and even used to make a traffic management decision. The proxy can also decide to terminate the SSL connection or act as an end point. For an SSL rehandshake request, the proxy checks the SSL session ID for the party it received the Hello request from. If the SSL session ID points to a previous session, then session keys for the proxy interception are

derived from the same secret data logged by the proxy.

6. **Strong SSL proxy authentication with forced SSL re-negotiation, F5 Networks:** The patent [28] reciprocates an idea based on the same design of Split TLS. In the setup, the client initiates an encrypted network connection with a proxy. Having established a secure connection with the proxy, it forward the target server domain name or IP address it wants to connect to. The proxy responds to the client with an encrypted session re-negotiation message. The client sends to the proxy an encrypted session handshake message. The proxy device forwards the encrypted session handshake message to the target server, and continues to pipe handshake messages between the client and the server device, enabling the client device and the target server device to establish an encrypted session.

7. **Authentication delegation based on re-verification of cryptographic evidence, Microsoft:** The patent [57] discusses on authentication delegation by re-verifying the cryptographic evidence. The client authenticates to the proxy (gateway) using a TLS handshake with client authentication. A recording of the TLS handshake (THR) is provided either to the web server (which re-verifies the validity of the handshake) or to a third party entity (which upon verifying the recording, provides user credentials to the gateway which is further authenticated with the web server). The web server or third party verify that the user has authenticated to the gateway (by validating the certificate verify message in the THR). If the user credentials (e.g., client certificate) are authenticated by the web server, access to the requested web server is granted to the client/user, and if the client certificate cannot be verified, access is denied. The scheme is limited to function only when there is client side authentication in

TLS and the client and gateway do not resume a previous TLS session/duplicate an existing session. Replay attacks to reuse the recorded TLS handshake record are prevented by using timestamps in handshake messages or embedding of a nonce (provided by the web server) by the proxy.

3.3 Sample evaluation based on the UDS benchmark

In this section, we discuss some of the specific cases and give an example for how some of the properties are rated.

1. ***Middlebox discovery visibility*:** An example of a scheme that is 'Yes' rated is TPS. The TPS RFC relies on a TLS extension which is used by the proxy to provide information to the client or server about its presence. They leave it to the client applications to visualize the existence of a proxy to the user. An example of a scheme that is rated as 'No' is Split TLS as a certificate authority. In this case, the middlebox acts as a CA. It can present a certificate (signed by a legitimate root certificate) to the client/user. The client/user would not be aware that the connection is intercepted. An example of a scheme that is rated as partially compliant is regular Split TLS. In this case, an user/client accepts a certificate for the middlebox during configuration. The user/client may or may not be aware if the connection is intercepted. All schemes that use a predefined list of middleboxes to include in an encrypted connection or need a predefined configuration to connect to the middlebox are rated as partially compliant.

2. ***Infrequent errors or open fails*:** An example of a scheme that is rated 'No' is Split TLS as in Split TLS, authentication mechanisms like key pinning

are not supported. An example of a scheme that is rated 'Yes' is TPS as they support other authentication mechanism like DANE, HSTS etc. A scheme that makes modifications by extending TLS does not support *Infrequent errors or open fails* as it is unclear how open fails are handled by the client and the server. A scheme in this case is rated 'No' unless they discuss specifically if they support existing alternative authentication mechanisms for TLS.

3. ***No significant latency*:** The number of handshakes in Split TLS for a single TLS connection are two (one each between a client and a middlebox and a middlebox and a server) and hence does not support *No significant latency*. Schemes are broadly categorized as to use upto three handshakes in their protocol. On a general note, three handshakes mean that there is one TLS handshake between the client and the server, one TLS handshake between the client and the middlebox and then one TLS handshake between the middlebox and the server. Schemes that use two TLS handshakes mean that generally there is one TLS handshake between the server and the middlebox and one between the middlebox and the client. Schemes that have one TLS handshake generally have some kind of an additional extension that is included along with the TLS Sever/ClientHello. This extension would carry information to include and authenticate a middlebox in the session. The entire TLS session is negotiated in two RTTs (round trip times; the same number of RTTs in a regular TLS connection with no middlebox).

4. ***Minimal read/write disclosure*:** A scheme is rated as partially compliant when the end points can control what components of the traffic can be read/-modified by the middlebox. An example of such a scheme is mcTLS.

5. ***Server compatibility*:** The schemes do not support *server compatibility* when

modifications are made to the standard HTTPS/HTTP protocols or involve the server to be modified for the middlebox to intercept the connection. An example of a scheme that is rated as 'No' is mcTLS.

3.4 Evaluation of schemes (client consent) based on the UDS benchmark

In this section, we evaluate all schemes that provide client side consent to the middlebox (grouped under their category). We choose and discuss the rating for one representative scheme in detail for each category. For the rest of the schemes in the category, we only discuss the differences in the rating process relative to the representative scheme.

3.4.1 Modified TLS

1. **mcTLS:** mcTLS complies partially with *Middlebox discovery visibility* as it uses a predefined list for middleboxes that the client and the server want to include in the connection. The scheme does not support *Middlebox persistence visibility* because for each connection, the client sends a list of middleboxes (included in the ClientHello) that it want to include in the connection. If an user switches a network, the client would still continue to ask the server to include the list of middleboxes in the connection. mcTLS fails to discuss how it would handle open fails or other authentication mechanisms and thus does not comply with *Infrequent errors or open fails*. mcTLS requires the server and middlebox to support the scheme and hence does not support *Server compatibility* and *Middlebox compatibility*. mcTLS achieves the entire TLS negotiation in one extended TLS handshake (here extended TLS handshake is based on the three

way key exchange protocol. It is a regular TLS handshake with symmetric key establishment for middlebox) and hence supports *No significant latency*. As described in the three way handshake of mcTLS, it is clear that the scheme supports *Server authentication* and *Middlebox recognition* and *TLS protocol negotiation*. An application and server using mcTLS can set contexts in two ways: context per data stream or as a middlebox policy. A middlebox can read/write data only if the it has the respective context key. Hence, we rate mcTLS as to partially support *Minimal read disclosure* and *Minimal write access* and support for *Connection specific interception*.

2. **EFGH:** mcTLS and EFGH are rated similarly as they both achieve the same properties using a subtly different approach. EFGH handles middlebox discovery and discusses very briefly that the client first establishes a TCP connection with the middlebox by a proxy auto-config file, DHCP, or manually configuring the address of a proxy in browser. An user can switch configuration on how he connects to the proxy by changing his browser settings, DHCP configuration etc. when he switches his network and is hence rated to support *Middlebox persistence visibility*. All other properties are rated the same as mcTLS for similar reasons.

3. **Blindbox:** mcTLS and Blindbox are rated similarly. Blindbox does supports *Minimal write access* as the middlebox cannot modify traffic between the client and the server. The scheme requires three separate streams: one regular TLS connection, one to transmit the "searchable" encrypted tokens, and one to listen if a middlebox on path requests garbled circuits. Only one TLS handshake is used between the sender and a receiver in the scheme. The performance overhead of Blindbox on a connection is due to obfuscated rule encryption and only one TLS handshake is used between the end points. We rate is as to support

No significant latency. All other properties are rated the same as mcTLS for similar reasons.

3.4.2 Passthrough

1. **Ubicrypt:** The scheme does not discuss how an user would be aware of the middlebox inspecting the packet payload. The trusted gateway (middlebox) in Ubicrypt is preconfigured to be a network hop in the connection and hence partially complies with *Middlebox discovery visibility.* The gateway is no longer part of the network and the client defaults to using a regular end-to-end TLS connection with the server when the user switches a network. It hence supports *Middlebox persistence visibility.* Ubicrypt would rely on existing browser alternative authentication mechanisms and hence supports *Infrequent errors or open fails.* Even though the real world performance for their proof of concept was poor, the protocol establishes a QUIC connection in one extended TLS handshake (extended TLS handshake here means the regular TLS handshake along with the connections that leaks the QUIC sessions keys) and hence supports *No significant latency.* Ubicrypt is deployable and supports *Server compatibility* as no server side modifications are required, however, does not have support for *Middlebox compatibility.* Ubicrypt supports *Server authentication* and *Middlebox recognition* as the scheme allows the client to authenticate the server using QUIC and the threat model only allows a trusted gateway as a middlebox. The scheme is modeled such that clients simply failing to leak the keys would have the network connections terminated before it sends QUIC application data packets to the server. The keys have to be leaked for each connection or the client does not communicate with the server. We rate it to support *Connection specific interception.* It is unclear if the MAC keys are also leaked along with

the encryption keys in Ubicrypt. We assume that for the gateway to correctly decrypt and verify integrity for the data, the MAC keys are leaked. Following this assumption, Ubicrypt does not provide for *Minimal read disclosure* and *Minimal write access*. Ubicrypt supports *Client negotiations for TLS* as the client communicates with the server directly.

2. **Triraksha:** The rating for Triraksha is described in Section 4.4.1.

3. **SRPK:** SRPK supports *Middlebox discovery visibility* as the client sends a list of middleboxes that are in connection to the server and allows for a unknown middlebox to be added to the connection as well. SRPK supports *Middlebox persistence visibility* as the client sends the hashes of the certificates of middleboxes that it knows are on-path to the server. If an user switches a network, the new connection to the server would include only the new list of middleboxes in the network. SRPK does not support *Server compatibility* as a new content type 'KeyshareInfo' is used in the TLS connection. SRPK does not support *Minimal read access* and *Minimal write access* as the client shares the RSA encrypted record of the write keys (for both client and server) with the middlebox. SRPK allows the client to have access controls for the proxy over a specific domain and hence supports *Connection specific interception*. SRPK establishes the connection in one TLS handshake and hence supports *No significant latency*.

3.4.3 Proxy

1. **Split TLS:** With Split TLS, the middlebox certificate is installed by the client upon first connection. This is a case of preconfigured middlebox and is hence rated as to partially support *Middlebox discovery visibility*. Once a certificate from the client is installed on the root certificate directory, the client/user is

always configured to allow the proxy in the connection. It does not support *Middlebox persistence visibility*. With Split TLS, authentication mechanisms like HSTS and key pinning are not allowed. It does not support *Infrequent errors or open fails*. No modifications are needed at the middlebox and server end and hence supports *Server compatibility* and *Middlebox compatibility*. Split TLS uses two handshakes (one between the client and the middlebox and one between the server and the middlebox) and hence does not support *No significant latency*. With Split TLS, the server is not authenticated by the client and the client does not negotiate the TLS handshake. It is hence not *Server authentication* and *Client negotiation for TLS*. The client can see the certificate of the Middlebox and is hence *Middlebox recognition*. Split TLS does not allow for *Connection specific interception*. Once a certificate is installed on the client, the middlebox is capable of modifying and reading all encrypted traffic in the connection. It hence does not support *Minimal read disclosure* and *Minimal write access*.

2. **Split TLS as a CA:** Regular Split TLS and Split TLS as CA are rated similarly. However, the difference lies in *Middlebox discovery visibility*. The client trusts the middlebox as the middlebox certificate is signed by the same root CA as that of a root certificate on the client computer. It is hence does not support *Middlebox discovery visibility*. The scheme does not support *Middlebox recognition* as the middlebox presents its certificate to the client signed by a CA that the client trusts. The rest of the properties are rated the same as Split TLS for similar reasons.

3. **Exproxy:** Split TLS and Exproxy are rated quite similarly. With Exproxy the client is preconfigured to connect to the proxy and is hence partially compliant with *Middlebox discovery visibility*. In Exproxy, an user can switch configuration on how he connects to the proxy by changing his proxy connection settings and

is hence *Middlebox persistence visibility*. Exproxy is not *Server compatibility* and *Middlebox compatibility* as it requires modifications at the middlebox and the server end. The trusted proxy model does not support *Minimal read access* as it has access to all the traffic in the connection while the threat model of using a caching proxy partially supports *Minimal read access* and supports *Minimal write access*. The caching proxy is authenticated and it is not provided with the decryption keys to the encrypted traffic. However, the caching proxy can still serve the client with cached data and hence is rated partially compliant. The rest of the properties are rated the same as Split TLS for similar reasons.

4. **TPS:** TPS supports *Middlebox discovery visibility* as a TLS extension is used by the proxy to provide information to the client or server about it. It supports *Middlebox persistence visibility* as the custom extension would be used to involve a middlebox for each connection irregardless of the network. TPS supports *Infrequent errors or open fails* as they specifically mention that the scheme can support alternative authentication mechanisms. TPS does not support *No significant latency* as it uses two TLS handshakes to make a connection. TPS is not *Server compatibility* and *Middlebox compatibility* as it requires the server and middlebox to support the 'ProxyInfo' extension. TPS does *Server authentication*, *Middlebox recognition* and *Client negotiation for TLS* in their scheme. These three properties are supported by the use of the 'ProxyInfo' extension. The extension is send with each connection and hence TPS is *Connection specific interception*. Once the proxy is part of the connection, it has access to all the traffic in the connection and is capable of modifying it. It does not support *Minimal read disclosure* and *Minimal write access*.

5. **ETP:** ETP is rated similar to Split TLS for most of the properties for similar reasons. ETP supports *Connection specific interception*, *Middlebox discovery*

visibility and *Middlebox persistence visibility* as it involves the user/client consent for each connection.

		Usability			Deployability			Security					
	Name	Middlebox discovery visibility	Middlebox persistence visibility	Infrequent errors or open fails	No significant latency (handshake)	Server compatibility	Middlebox compatibility	Server authentication	Middlebox recognition	Connection specific interception	Minimal read disclosure	Minimal write access	Client negotiations for TLS
Proxy	Split TLS	◐	○	○	○	●	●	○	●	○	○	○	○
	Spit TLS as CA	○	○	○	○	●	●	○	○	○	○	○	○
	ETP	●	●	○	○	●	○	○	●	●	○	○	○
	TPS	●	●	●	○	○	○	●	●	●	○	○	●
	Trusted Exproxy	◐	●	○	○	○	○	○	●	●	○	○	○
	Caching Exproxy	◐	●	○	○	○	○	○	●	●	◐	●	○
Pass through	SRPK	●	●	●	●	○	○	●	●	●	○	○	●
	Ubicrypt	◐	●	●	●	●	○	●	●	●	○	○	●
	*Triraksha	◐	●	●	●	●	○	●	●	●	○	●	●
Modified TLS	mcTLS	◐	○	○	●	○	○	●	●	●	◐	◐	●
	Blindbox	◐	○	○	●	○	○	●	●	●	◐	●	●
	EFGH	◐	●	○	●	○	○	●	●	●	◐	◐	●

Table 1: Summary of the evaluation of client side consenting schemes with the UDS benchmark. Legend: ● the scheme supports the property; ○ the scheme does not support the property; and ◐ the scheme partially complies with the property

3.5 Related work omitted from UDS framework

In this section, we discuss some of the related literature that is not included with the evaluation in the UDS framework. These proposals are not included with the UDS

framework as they either lack sufficient technical details or are only partially related to our work.

Stebila et al. [75] examines the security and performance of CloudFlare's Keyless SSL, in which the principal web server retains possession of their private key and splits the TLS state machine geographically with the edge server and a key server. Keyless SSL allows to use a private key proxy service and hence, decreases the trust given to an edge server while still being able to do data caching and compression. They find the latency to be sightly higher but better than the principal web server serving content directly to the edge server.

RFC 3280 [78] specifies the Internet standard for proxy certificates based on the X.509 public key infrastructure. The document defines properties for proxy certificates as a means of providing restricted proxying within an (extended) X.509 PKI based authentication system.

Any node refusal [1] includes proxies in TLS sessions by requiring trusted proxies to have their own end-to-end HTTPS session with the browser when processing traffic from an HTTPS content server. The proxy establishes an end-to-end intra-connection using TLS with a NULL cipher (A cipher with no encryption but still does data authentication and data integrity checking) between the browser and the server. The traffic sent from the client to the server is encrypted by the end-to-end TLS sessions, while being viewable by the proxy. The browser and web server can authenticate and validate the integrity of the data. Any node (client/server) can refuse participation of sending data on a per object basis. The scheme is not included in the UDS framework due to lack of technical detail in the paper.

Several patents [29], [46], [57], [35] and [54] discuss methods like optimizing SSL handshakes, support for delegation of SSL handshakes to secondary server devices, form based login authentication in SSL through a proxy and insertion of resources by

a proxy in Split SSL. Most of these ideas are represented and similar to some of the patents in the previous section and hence we do not discuss them.

Liang et al. in their paper [52] study the current practices of using HTTPS with CDNs. They investigated 20 popular CDN providers and 10,721 of their customer web sites. They observed that 15 percent of them raised alerts of invalid certificates, reifying the broken trust model of HTTPS with CDNs. They also provide a lightweight and flexible DANE based solution that addresses the HTTPS authentication problem in the CDN environment. In the solution, the client issues a DNS query request to view the web server's TLSA records and can recognize the delegation relationship between entities.

Jawi et al. [45] use a non intrusive, forward proxy with adaptive security features for SSL/TLS connections. The proxy has three components for monitoring, analysis and response for each connection. A security policy is used with the analysis component which feeds information of each SSL/TLS connection for static and dynamic testing before proceeding to the response component. The security policy contains static attributes (cipher suites, root certificates, key exchanges, ciphers and hashes) and dynamic attributes (SSL certificates specific to a connection). The JSON schema is used to define the attributes and policies used for testing. Scripts are then run to match the attributes with the policies and if the attributes default to the policy then an error is raised in the response component.

RFC 2660 [65] defines Secure HTTP (S-HTTP) which provides secure communication mechanisms between an HTTP client-server duo. They aim to improve the status-quo for commercial transactions in a wide range of applications. S-HTTP does not require client side public key certificates and supports only symmetric key operation modes. S-HTTP defines two mechanisms for key transfer, the first uses

"public-key enveloped key" exchange and the second method uses "externally arranged keys". For the latter method, the data is encrypted by using a prearranged session key, with key identification information specified in the header lines. The S-HTTP protocol also specifies interaction between a client and a proxy in their protocol. The client and a proxy negotiate cryptographic options that the proxy sent. On agreement, a client will recursively encapsulate the data and send it to the proxy. When the proxy receives a message, it will strip the outer encapsulation to recover the message and pipe it forward to the server.

Backes et al present WebTrust [26], an integrity and authenticity framework for HTTP that "allows on the fly verification of static, dynamic and real time web stream content from untrusted servers". It uses iframes for verifiability of content provided from different web servers. Under their model, WebTrust provides protection of HTTP content against active network attackers. WebTrust enables the client to detect any modified data packet upon arrival without downloading the entire document. Webtrust also enables the use of web cache (proxies) but looses confidentiality of content in this case.

Some proposals work to incorporate middleboxes by working on a different layer in the OSI stack. IPsec [71] proposes to encrypt and authenticate data at the IP layer. There are significant limitations to use IPsec with middleboxes. The overhead of double encryption when TLS is used with IPsec and the lack of separate key management scheme make the adoption less likely. TCPcrypt [18] establishes end-to-end encrypted sessions at the TCP layer. TCPcrypt has reduced overhead on the server and leaves authentication to the application. However, TCPcrypt does not replace TLS and suffers from drawbacks like: using short lived keys to provide forward secrecy and the lack of a key confirmation in its 4 way handshake. The Delegation-Oriented Architecture and (DOA) [3] and Named Data Networking (NDN)

[8] projects support to use trusted intermediaries in the network based on their own security mechanisms and properties.

3.6 UDS evaluation discussion

We summarize our evaluation of the proposals in Table 1. It is observed that within the schemes there is inherent lack of consideration in the design for usability and deployability. Rendering the user to believe that it is is communicating with the end server directly when infact it is communicating with a middlebox results in violation of trust for the user. The schemes would barely describe the bare bones on how a deployed scheme would look like to an user. A second observation is that almost all schemes requires modifications at the server, client and middlebox end points. Non server compatibility results in overall less integration of the scheme in the real world. Requiring server side modifications forces clients to connect to only servers that support the scheme. Some schemes fail to offer fine-grained security within the middleboxes. While some schemes by design apply minimal read and write disclosure to the middlebox, a majority of them do not discuss it in detail. The evaluation for majority of the schemes are limited. Some do not have a proof of concept implementation and the subset of the ones that do have an implementation are not evaluated extensively over real world data points. However, the reason for this may be attributed as evaluation of a scheme on the Internet is not trivial. There are limited ways to demonstrate and test ideas on a large scale when there is already a huge install base that is reluctant to change.

In this thesis, we do not do a quantitative scheme evaluation, that is weights are not assigned for each property. Simply counting the dots from Table 1 do not endorse a scheme to be better. However, from our perspective the most important property

within the framework is *Server compatibility*. As with existing literature, if a scheme requires server side changes, the chances of it being deployed in the real world is less likely. If we look at Table 1 and in particular at the column for server compatibility; only 5 schemes are server compatible. Two schemes (Ubicrypt and Triraksha in theory present the same idea and differ slightly in the level of fine-grained confidentiality, Further, Ubicrypt was developed for the QUIC protocol and we propose Triraksha for the TLS protocol). The remaining three schemes collectively lack the necessary security properties and break end-to-end encryption for reasons described in Section 2.2.3. This gives us incentive and the need for a scheme that provides fine-grained confidentiality and is server compatible.

We now propose and introduce the Triraksha protocol in Chapter 4.

Chapter 4

Triraksha

In this chapter, we discuss and set the context for our scheme: Triraksha. With Triraksha, we can serve end-to-end encrypted traffic in a TLS connection while maintaining middlebox functionality. We use software modifications with TLS to enable fine-grained security and usability at end points for incorporating middleboxes with TLS. Triraksha allows the middlebox to access plain text content for a TLS connection but does not allow for modifications to the traffic. In the subsequent sections, we describe our protocol requirements, key concepts, and the protocol itself. We follow up with the evaluation in the last section of this chapter.

4.1 Design goals

With Triraksha, we would like to have the security properties offered by TLS with fine-grained confidentiality for middleboxes. Further, our priority is to build a scheme that can be easily deployed with existing infrastructure. We target the following in Triraksha:

1. Entity authentication: A client should be able to authenticate the server and the middlebox in a TLS connection. The connection used for communicating

data between each entity in a TLS session must be end-to-end encrypted. The encryption ciphers used in TLS should not be weak.

2. Payload confidentiality: Any party other than the end points should not be able to read the plain text send between the client and the server unless it is given consent by the client or the server.

3. Payload integrity: Clients should be able to verify the data is sent by the server it is communicating with. Integrity of data should be maintained by end points and modification to data should be detected by the end points.

4. Configurations and deployment of middleboxes should be on par with recent practices and be *Server compatible*. There should be minimal change to existing protocols/software in order to maximize deployment for the scheme.

5. The middlebox should not be able to modify the traffic. It should comply with *Minimal write access*.

6. Read/write privileges for a middlebox on encrypted traffic should be connection specific and should be set by the user. An user should be able to opt out of the communication channel if he would not like a middlebox to inspect his traffic.

7. The user should be aware of the middlebox in the connection.

In summary, we would like Triraksha to be compatible with existing infrastructure using minimal software/hardware modifications. The properties should incorporate and satisfy the security offered by TLS and give privileges for read permissions and no privilege for write permissions to the middlebox for a connection. Atleast, one of the end points should authenticate the middlebox and the user should be aware of the middlebox in the connection.

4.1.1 Threat model

Scheme and third party attacker: Triraksha relies on the security properties of TLS for an encrypted connection between a client and server. All entities in the scheme execute the protocol correctly and do not leak information to an entity outside of the scheme. Triraksha relies on the browser/TLS library for existing TLS authentication techniques to detect an untrusted connection (through the existing browser warning signs). A third party adversary in our scheme is an active network attacker. He is computationally bound and can attempt to intercept, alter, or insert packets during any phase of the session. Our scheme does not defend against denial of service attacks and side channel attacks on network protocols. The adversary does not have access to private keys of servers. The server and the client do not collude in any way.

Middlebox and user: The middlebox can attempt to modify or read the plaintext from the encrypted traffic in the end-to-end connection. The middlebox trusts the encryption keys provided by the client. A 'malicious user' can send garbage TLS encryption keys for a connection to the middlebox. The middlebox trusts the encryption keys provided by the client for a given grace period until it is provided with the master secret used in TLS negotiations (see Chapter 5 for details). A 'malicious/defaulting user' can attempt to bypass the Triraksha protocol.

4.2 Triraksha overview and architecture

Overview: To achieve the requirements of Triraksha, we allow clients to connect to a server only if they provide the middlebox with the TLS secrets for that connection. We define TLS secrets as the symmetric session encryption keys/IVs that are used to encrypt/decrypt data packets for that specific TLS session. Individually sharing a

TLS encryption key for a session implies that the middlebox can inspect TLS traffic for that session. Further, the client can communicate effectively with a server only if it provides the TLS secrets. In our scheme Triraksha, we are interested in application layer middleboxes that are capable of inspecting the traffic but cannot modify it. Application layer middleboxes are typically implemented at a common gateway for all clients in an organization's network. The gateway typically implements application layer functionality like packet inspection, content filtering, audit logging etc. In large organization networks, a gateway can be a network wide proxy. In our scheme, our middlebox is a gateway which implements a packet inspection service on an TLS connection between a server and a client.

Example use cases: The concept our systems presents is best put to use in enterprise environments. In an enterprise environment, the need to inspect TLS traffic is essential to mandate network policies. The Triraksha middlebox can act as a corporate firewall without having clients install root certificates. An IDS/IPS device that currently ignores encrypted traffic or relies on Split TLS can now have access to the plain text in a TLS connection by adopting Triraksha. Institutions and home devices can use Triraksha as a parental filtering device to block inappropriate content. On a general note, Triraksha can be adopted for application layer middleboxes that require only read access in a TLS connection. Following the examples (demonstrated in mcTLS [59]) for middleboxes that require read access in connections, we picture the Triraksha middlebox to be used as a load balancer, packet pacer and WAN optimizer.

In Triraksha, we only give read permissions to the middlebox. A majority of the middleboxes which provide functionality over HTTP/TCP messages between end points only need read access to the request/response headers and body [59]. Further, if a middlebox has write permissions then there is no way to maintain integrity for

the data with the current HTTPS protocol. Hence, with Triraksha we disallow write access to the middlebox. The Triraksha middlebox cannot be used as a data cache or data compressor. TLS uses MACs for content integrity between the client and the server. The MACs are specific to a client-server connection, therefore no caching of HTTPS content is possible without significant changes to the way HTTPS works. With Triraksha, deployability is one of our design goals and we do not make changes to HTTPS. We suggest the reader to look at SSL Splitting [51] for further discussion on this.

Architecture: Figure 3 represents the architecture for Triraksha. Triraksha serves end-to-end encryption using TLS between a client and a server while still allowing the middlebox to inspect packet payload. The Triraksha middlebox initially allows any client to communicate TLS handshake packets and TLS cipher key exchange packets with the server but drops TLS application data packets. Selective blocking of TLS packets is done at the network level by firewall rules. With user consent and successful completion of a TLS handshake, the client leaks the TLS encryption keys for a connection to the middlebox in a separate secure connection. On receipt of the TLS encryption keys, the middlebox will allow TLS application data packets to pass for the respective connection. The client and the server communicate with each other like in regular TLS for that TLS connection while the Triraksha middlebox uses the encryption key to decrypt the encrypted traffic and gets access to the plain text. This allows the middlebox to enforce its inspection policies while still maintaining a sense of end-to-end encryption between the client and the server.

Compared to Split TLS where once a root certificate is installed and the middlebox has complete access to the traffic for all the connections, Triraksha offers fine-grained security. We would like to note that the trust level between the client

and the Triraksha middlebox is stricter than the trust between the client and the Split TLS middlebox. Specifically in Triraksha, the client controls for which connection the middlebox can have access to the plain text by sharing only the encryption key for the TLS session. Further, the middlebox can never modify the traffic. In Triraksha, the trust on the middlebox is minimized but not null. While middlebox discovery is out of scope in this thesis, we assume that the user/client is aware of the middlebox and explicitly provides trust/authenticates the middlebox in a separate connection. Specific to the Triraksha implementation, the user may learn the public key/FTP/SSH password of the middlebox out-of-band. For example, for a corporate network, the public key/password for the middlebox connection might be given on a piece of paper or in an email.

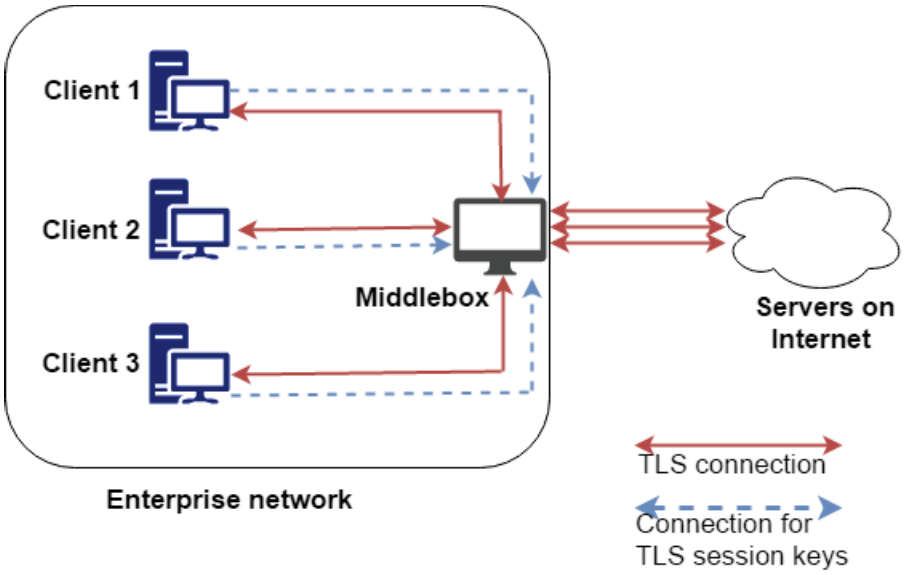

Figure 3: Architecture of the Triraksha protocol.

On a design level, the Triraksha protocol is faster than Split TLS. Split TLS requires two TLS handshakes to perform to establish the connection between the client and the server, while the Triraksha protocol needs only one TLS handshake to establish a connection between the client and the server. The connection between the client and

the middlebox for sending keys is long lived and needs to be established only once.

Once application data is flowing, Split TLS has to do an encryption and decryption for every record. Specifically, the middlebox first decrypts the data sent by the client, performs its functionality and then re-encrypts this data to be send to the server. Similarly these operations are performed for messages sent from the server to the client. With Triraksha, the decryption operation (where the middlebox get access to the plain text) is non halting. The decryption is performed for encrypted data logged by the client and does not actively block application data packets for the connection.

We now proceed to explain how our scheme functions by discussing the handshake phase and the record phase of Triraksha in detail.

4.2.1 Triraksha handshake protocol

The Triraksha handshake is similar to the TLS handshake and can make use of all the cipher suites and extensions associated with TLS.

The Triraksha handshake is initiated by the client and allows it to:

1. Authenticate the server.

2. Negotiate a symmetric session key and a cipher suite with the server.

3. Allow the user to leak TLS secrets to a middlebox.

Figure 4 represents the Triraksha handshake.

Handshake steps: As in TLS, the Triraksha client opens a TCP connection and sends a TLS ClientHello message. The ClientHello message contains the following: (1) the maximum protocol version that the client wishes to support (2) the 'client random' (32 bytes, out of which 28 are suppose to be generated with a cryptographically strong number generator) (3) the 'session ID' (in case the client wants to resume a session

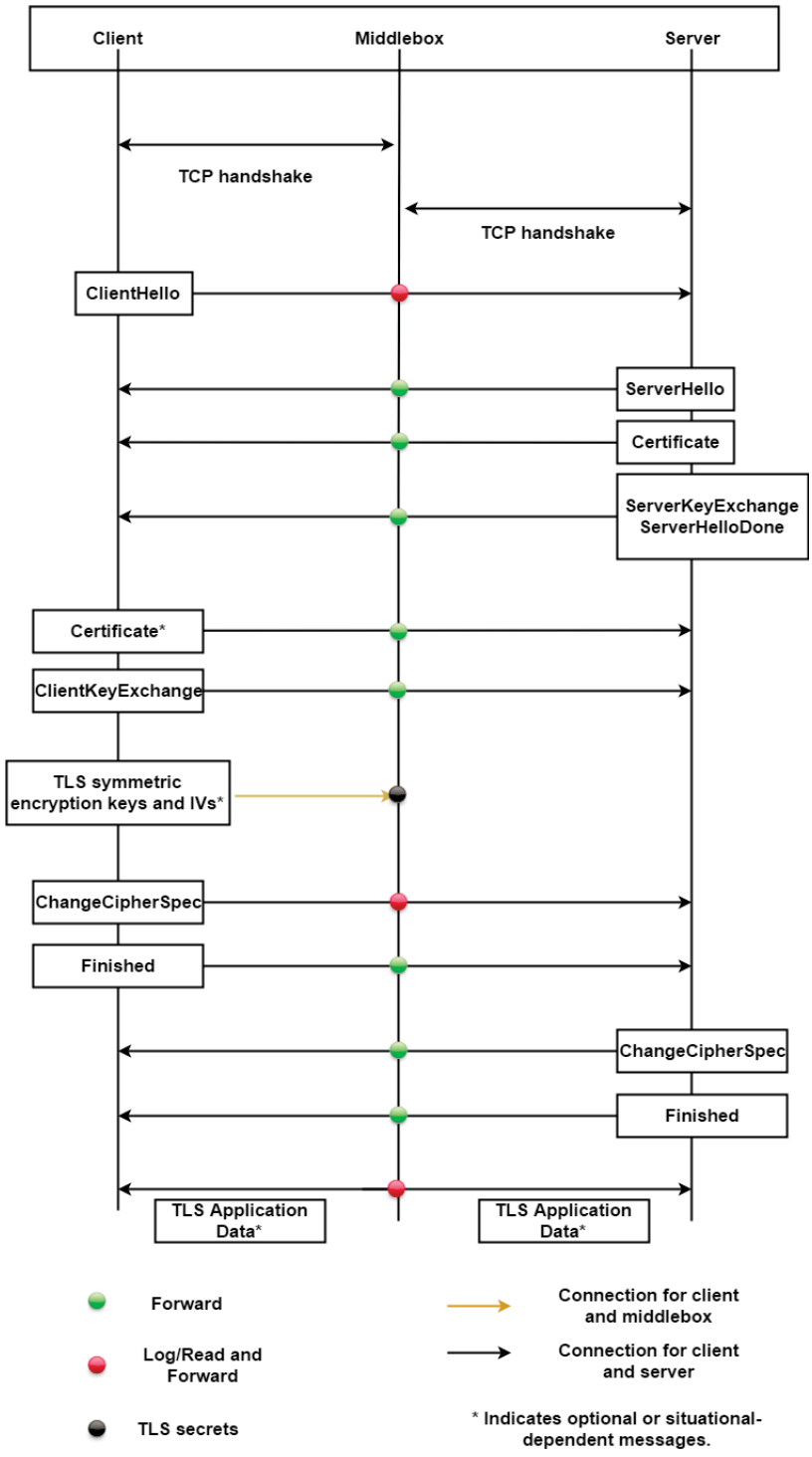

Figure 4: Triraksha handshake.

in an abbreviated handshake) (4) the list of 'cipher suites' that the client supports, ordered by client preference (5) the list of compression algorithms that the client knows of, ordered by client preference (6) some optional extensions.

The middlebox simply forwards the ClientHello to the server and logs the client random, the source IP address, source port and destination IP address of the ClientHello packet. The client random is treated as unique to the TLS connection. For conciseness in the thesis, we refer to this process of logging the client random as 'Middlebox ClientHello dissection'. The format for the log file described above is:

XXXXX	source_IP_address	source_port	destination_IP_address

XXXX is the client random in hex value of the ClientHello

Figure 5: Format of log file from Middlebox ClientHello dissection.

The server responds to the ClientHello with a ServerHello which contains the following: (1) the protocol version that the client and server will use (2) the 'server random' (32 bytes, with 28 random bytes) (3) the session ID for this connection (4) the cipher suite that will be used (5) the compression algorithm that will be used (6) optionally, some extensions. Along with the ServerHello, the server also sends its Certificate which contains the server's public key, the ServerKeyExchange which contains the server's value for key exchange and a ServerHelloDone. Following the ServerHelloDone, the client responds with the ClientKeyExchange which is the client part of the key exchange. Using the data from the key exchange messages, the client and server establishes a pre master secret, which is further used to generate a master secret. The master secret is used in combination with the client random and server random and fed to a pseudo random function to generate a key block. The key block is split into sets of keys: encryption keys, MAC keys and IVs. The keys are used in

the subsequent messages to encrypt/decrypt data. The client also sends a ChangeCipherSpec message, which is a separate TLS record type. The ChangeCipherSpec marks the point at which the client switches to the newly negotiated cipher suite and keys. The subsequent records from the client will then be encrypted. The client sends a 'Finished' message, which is a cryptographic checksum computed over all previous handshake messages (from both the client and server). When the server receives that message and verifies the checksum, the server obtains a proof that it has indeed communicated with the same client all along. The 'Finished' message protects the handshake from alterations by a third party. The server responds with its own ChangeCipherSpec and Finished messages at which point the handshake is complete.

In Triraksha, the user decides if he wants a middlebox to inspect traffic for a TLS connection. He makes the decision before attempting to connect with the server. Following the decision to support packet inspection, the client logs the sets of keys generated from the key block during the key exchange for each TLS connection made by the client. Specifically, the data logged is a set of encryption keys and IVs. In Triraksha, the MAC keys are not logged and/or shared with the middlebox. The client uses the client random from the ClientHello message as a unique identifier and associates it with the keys. From now on, we refer to a set of client random and TLS symmetric encryption keys and IVs for a connection as the TLS secrets for that connection. The TLS secrets for the respective connection are stored on the client computer. The client authenticates the middlebox in a separate connection and leaks the TLS secrets to the middlebox on user consent. The client can authenticate the middlebox using any secure authentication process like certificate validation etc.

To summarize, the Triraksha handshake achieves the following:

1. The client authenticates the server.

2. The client does the TLS handshake negotiations with the server. Specifically, the client establishes a symmetric encryption key with the server for encryption of TLS record packets.

3. The client authenticates the middlebox.

4. The user consents to leaking TLS secrets for a connection and following the decision, the client leaks the TLS secrets to the middlebox in a separate channel.

4.2.2 Triraksha record protocol

We now describe the additions we make to the TLS record protocol. Like in the handshake phase, the additions are external to the TLS record protocol in itself.

The Triraksha client takes payload from a higher level protocol and sets up an environment for encrypting/decrypting the data. The client breaks the payload into blocks. Like in regular TLS, a payload in a block can consists of at most 16384 bytes. The client optionally compresses the block and then depending on the cryptographic primitives in the cipher suite, it applies a MAC and encrypts the block (the block is MAC and encrypted with the symmetric MAC and encryption key from the handshake phase). Triraksha supports use of any encryption and MAC algorithms that are supported by TLS. The encrypted block is appended with a TLS record header and transmitted to the end point (server) as a TLS application data packet. After reception at the other end point, the same operations are performed to the packet in reverse order (decryption, MAC verification, decompression, reassembly and sends it to a higher layer).

In Triraksha, the middlebox drops all TLS application data packets sent by the client until the client leaks the TLS secrets to the gateway for that TLS session. To

send TLS application data packets, the user would have to consent to leak the TLS secrets for a TLS connection. Following the decision, the client would have leaked the TLS secrets during the handshake phase.

The middlebox matches the client random in the TLS secrets it received by the client with the client randoms logged during 'Middlebox ClientHello dissection'. A successful match implies that the middlebox received the TLS secrets for a TLS connection and it should allow TLS application data packets to pass through for that connection. The middlebox now adds a firewall rule for the respective IP addresses and port (logged from 'Middlebox ClientHello dissection') and allows TLS application data packets to pass through for that connection. A client and server now continue to transmit TLS application data packets as in a regular TLS connection.

The middlebox uses the TLS secrets to decrypt the respective TLS application data packets to get access to the plain text. The middlebox however, cannot modify the data as it does not have the MAC keys for the TLS connection. This also implies that the middlebox must trust the secrets provided by the client. We discuss an extended threat model in more detail in Section 10.

To summarize, the Triraksha record protocol achieves the following:

1. Allows a client and a server to communicate TLS application data packets.
2. Provides read access to the middlebox in the TLS connection.
3. Provides no write privilege to the middlebox in the TLS connection.
4. The client and server can authenticate and maintain integrity of the TLS application data packets transmitted between them.

4.3 Implementation setup and discussion

The section discusses the setup for the implementation, the software modifications and implementation details for Triraksha. We discuss the modifications made at each end point based on the design decisions from the previous section.

A proof of concept for Triraksha was implemented on two computers running Ubuntu 14.01. Figure 6 represents the Triraksha implementation setup. The middlebox is set up such that it can connect to the Internet on a network interface (NI1). The client computer is connected to the proxy using a USB to ethernet adapter and communicates with the proxy on the respective network interface (NI2). The network interfaces NI1 and NI2 are bridged on the proxy computer hence, giving the client access to Internet. We describe how the client and the middlebox are implemented to support the scheme. Our implementation requires no server side modifications and should be compatible with all servers supporting TLS.

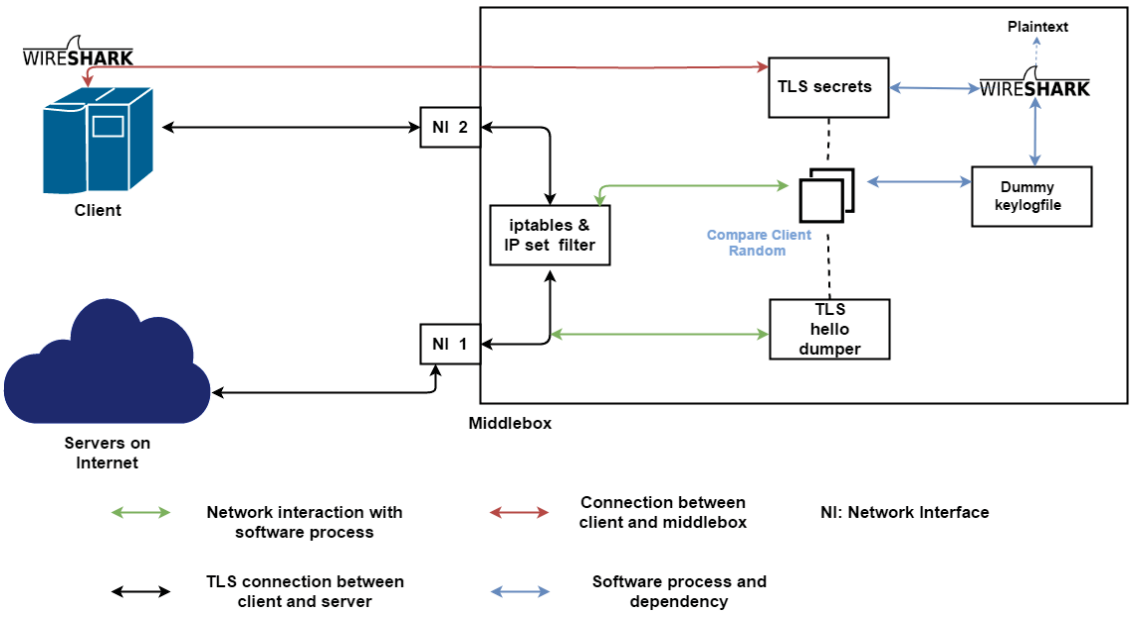

Figure 6: Triraksha implementation.

[1]Machine configurations described in the Appendix.

4.3.1 Client support for Triraksha

To enable Triraksha on the client end, we make software modifications such that the client computer can leak the TLS secrets to the gateway securely. On the client computer, TLS session key logging was achieved by the use of the SSLKEYLOGFILE variable which is supported by most popular browsers and SSL libraries (like Chrome, Firefox, OpenSSL, mbedTLS, GnuTLS etc). The SSLKEYLOGFILE variable sets up an environment such that every time an application makes a TLS connection, the underlying SSL library writes the TLS session master secret/pre-master secret to the environment variable location. The format for the SSLKEYLOGFILE is a series of records with the following formats [20]:

1. 'RSA xxxx yyyy'

   Where xxxx are the first 8 bytes of the encrypted pre-master secret (hex-encoded) and yyyy is the cleartext pre-master secret (hex-encoded).

2. 'RSA Session-ID:xxxx Master-Key:yyyy'

   Where xxxx is the SSL session ID (hex-encoded) and yyyy is the cleartext master secret (hex-encoded).

3. 'PMS_CLIENT_RANDOM xxxx yyyy'

   Where xxxx is the client random from the ClientHello (hex-encoded) and yyyy is the cleartext pre-master secret (hex-encoded).

4. 'CLIENT_RANDOM xxxx yyyy'

   Where xxxx is the client random from the ClientHello (hex-encoded) and yyyy is the cleartext master secret (hex-encoded).

In Triraksha, we do not share the negotiated TLS master secret or TLS pre master secret with the middlebox as it can be used to generate the encryption keys and the

MAC keys. To enable fine-grained confidentiality, we share only the encryption keys and not MAC keys. However, the SSLKEYLOGFILE does not provide a format to do so. A workaround for this is to modify an underlying TLS library or browser source code to print out the TLS secrets. However, by making this implementation choice we minimize adoption as the scheme would be adopted only if the modified library/browser would be used by the enterprise.

Instead of making changes to a browser/TLS library, we make use of Wireshark [20], a de-facto network protocol analyzer. In Triraksha, we use it to exfiltrate TLS secrets for a TLS connection. Wireshark is set up such as to capture network packets on the client computer and log all TLS packets for a connection. Wireshark requires the negotiated TLS master key for a connection to generate the TLS session keys (TLS secrets). The TLS master key for a connection is provided via the SSLKEYLOGFILE environment variable. The Wireshark TLS dissector waits for the handshake to complete and calculates the TLS sessions keys once it logs a ChangeCipherSpec message from the client end. The Wireshark TLS dissector was modified to print the TLS secrets to a file on the client computer. It should be noted that the ClientHello random, ServerHello random and master secret are not always sufficient to calculate the session keys. According to [27] RFC 7627 TLS Extended Master Secret extension, one needs to log the full handshake to generate the secrets and hence Wireshark waits until a client ChangeCipherSpec message is logged.

An user can control if he wants to leak the TLS secrets for a connection/all connections by simply exporting the client application with the SSLKEYLOGFILE variable. A script is run to leak the keys to the middlebox for each time Wireshark prints the TLS secrets to the file. The Triraksha client opens a connection and sends the file using ‘rsync’ over ‘SSH’ [14]. rysnc is a fast and versatile file copying tool. It can

copy locally, to/from another host over any remote shell, or to/from a remote rsync daemon. It uses the delta-transfer algorithm, which reduces the amount of data sent over the network by sending only the differences between the source files and the existing files in the destination. The rsync connection authenticates the middlebox. Other implementation choices that we could use for authentication between the client and the middlebox are to use a TLS client-server/SCP connection.

4.3.2 Middlebox support for Triraksha

To enable the middlebox to support Triraksha, the following need to be implemented:

1. Selective blocking of TLS packets on the network.
2. Receive TLS secrets from client and verify if they are associated with a legitimate TLS connection made by the client.
3. Decrypt encrypted traffic using the TLS secrets.

4.3.2.1 Selective filtering of TLS packets

Selective filtering of TLS content type is done on the network level with the help of iptables [5] and IP sets [4]. iptables is a Linux based firewall and is used to set up, maintain, and inspect the tables of packet filter rules in the Linux kernel. We leverage iptables with the u32 module to achieve filtering of TLS handshake and cipher key exchange traffic from TLS application data packets. We also note that it is possible to drop TLS application data packets using the string module for iptables. However, this may result in false positives as the string module compares a particular string anywhere in the packet. The u32 module allows for more fine-grained pattern matching as it allows one to check packet payload at a particular position in the packet bytes. We use u32 to check for TLS headers in the first few bytes of the TCP

payload. With the u32 module, we offset to the TCP payload and check the TLS header of a packet. TLS application data packets have the header value 17. The header value is followed by the TLS version used for the connection. The value for TLS 1.2 is 303. The following command is used to block application data packets for TLS 1.2.

```
iptables -I OUTPUT 1 \ -p tcp \! -f --dport 443 \ -m state
--state ESTABLISHED -m u32 --u32 \
"0>>22&0x3C@ 12>>26&0x3C@ 0 & 0xFFFFFF00=0x17030300 -j
   DROP
```

IP sets is used to set up, maintain and inspect so called IP sets in the Linux kernel. With IP sets we can store multiple IP addresses and port numbers and match against the collection by iptables as a whole. IP sets helps eliminate the performance penalty faced by iptables when adding IP addresses dynamically. With IP sets only one match rule is added to iptables instead of cluttering the firewall with multiple rules. IP sets prevents duplicate rules from being added by default. With this setup the middlebox allows all TLS handshake, TLS cipher key exchange and TLS alert traffic but blocks TLS application data packets for a TLS connection.

4.3.2.2 Associating TLS secrets to a TLS connection

To check if the TLS secrets received from the client are associated with a legitimate TLS connection made by the client, we build a program called 'TLS hello dumper' that uses libpcap to record and dissect TLS traffic, in particular the Hello messages in the TLS handshake. We log for each ClientHello message the client random, the source IP address, source port and destination IP address of the ClientHello packet. On subsequent and successful completion of the TLS handshake, TLS secrets are generated by the client computer and are transferred to the middlebox using rsync.

The format in which the TLS secrets are send to the middlebox is as follows:

```
Client Random XXXXX

Client encryption key YYYY

Server encryption key YYYY

Client IV YYYY

Server IV YYYY


where XXXX and YYYY are respective values.
```

The client random is used as a unique identifier to associate the secrets for a TLS connection. The TLS client random is a random string of 32 bytes unique to each TLS connection. The client and the server are allowed to exchange TLS application data packets only if there is a client random with the TLS secrets that matches with a client random logged by the middlebox. A script running on the middlebox matches the logged client randoms (from TLS hello dumper) with the client random presented with the keys sent by the client. This process allows the middlebox to link a TLS connection with the TLS secrets provided by the client for that particular TLS session. If a successful match occurs, it allows traffic for that pair of IP addresses and respective port to pass through. We track the TCP state of the TLS connection by inspecting the TCP headers of the packets. We remove a rule from the firewall when we detect a packet having a TCP FIN/TCP RST header in that connection. A single TCP connection encapsulates a single TLS connections. When a TLS connections ends, we remove the rule and by doing this we stop a client from attempting to initiate a TLS connection over a previously added firewall rule.

4.3.2.3 Decrypting TLS traffic at the middlebox

In this section, we first discuss our current implementation choices and then follow up with some alternative implementation decisions that could have been made.

To enable the middlebox to get access to the plain text from the encrypted traffic, the Wireshark TLS dissector for Wireshark version 2.2 was modified and run on the middlebox to read the TLS secrets received from the client. On successful read, Wireshark would decrypt TLS application data packets sent between the client and the server. Wireshark by design uses the SSLKEYLOGFILE to consume a master secret for a TLS connection and then generate TLS session keys. This process is done whenever it detects a TLS ChangeCipherSpec message for that connection. In our design, we feed Wireshark a dummy SSLKEYLOGFILE (a file that contains a fake master secret for each client random logged by the middlebox). An entry is added to the dummy SSLKEYLOGFILE only if the middlebox received TLS secrets for that connection. Whenever a client cipher key exchange packet is detected, Wireshark attempts to consume the master secret from a dummy SSLKEYLOGFILE. If Wireshark is unsuccessful in finding the master secret, it will continue to poll the dummy SSLKEYLOGFILE for 0.8 seconds (we set the threshold time to generate keys on the client and send it to the middlebox as 0.8 seconds). On successful read of the fake master secret, Wireshark generates TLS session keys using the fake master secret. These session keys are then overwritten by the TLS session keys that were sent from the client as TLS secrets. They are loaded into Wireshark process memory as a decoder for the rest of the TLS session. On subsequent exchange of TLS application data packets, the respective decoder is used to decrypt all TLS application data packets. MAC verification for packets are set to be ignored during this time.

The implementation effort involved writing multiple scripts (900 LoC) to enable functionality for our protocol and understanding and modifying parts of the Wireshark TLS dissectors which is written over 10,000 LoC. Another possible implementation choice that we could have made is to write a custom dissector that waits for the TLS secrets from the client and interprets the TLS record layer separately before the Wireshark TLS dissector is invoked. With our current design, we minimize the changes made to the Wireshark TLS dissector and attempt to load the secrets into memory exactly once when the cipher key exchange happens. Specifically, Wireshark uses a decoder variable to decrypt TLS traffic for a session. The secrets are loaded into the decoder variable when a cipher key exchange is logged by Wireshark. Once the decoder is set, it is used to decrypt application data packets for the entire TLS session. The decision to let Wireshark read a dummy SSLKEYLOGFILE and generate keys before overwriting them is because we wanted to make minimum changes to the dissector code. It is possible to load the keys from the client into the decoder variable directly by simply bypassing and commenting out the irrelevant code. However, the time taken for this step to read a dummy keylog file and generate secrets is negligible (in ms). The process of reading a SSLKEYLOGFILE provides for a neat GUI functionality to be used at a later stage for determining integrity of the TLS keys and data. Specifically, if we would like to verify the integrity of the payload (MAC verification) and the integrity of TLS encryption keys, the client must submit his end of the SSLKEYLOGFILE (which contains the correct master secret). The client SSLKEYLOGFILE can then be loaded via the GUI and MAC verification can be performed for the TLS application data packets.

We also note that there are a number of potential implementations for the Trirakksha middlebox. We explore a number of potential projects like using NFQUEUE [10], Scapy [15], nDPI [9], mitmproxy [7] and Surciata as an IPS. However, with most of

these projects we would have required to change underlying libraries (which would result in decreased deployability) or account for TCP and TLS reassembly. We choose Wireshark as it a de-facto network protocol analyzer with state of the art implementation for TCP, TLS reassembly and TLS decryption. Our current implementation is portable to existing TLS libraries and browsers. Ubicrypt is the only other scheme in the literature to have a proof of concept which can be adopted with web browsers. However, even their proof of concept was implemented over a virtual network.

4.4 Evaluation

To evaluate Triraksha, we aim to understand the following:

1. Does Triraksha function correctly?

2. What are the performance overheads for Triraksha?

We begin by first understanding the goals achieved by Triraksha, follow up with functionality evaluation and finally end with evaluating the overhead of using the scheme when compared to a regular end-to-end TLS connection.

4.4.1 Design principle compliance

To understand how Triraksha achieves its design goals and fares against other schemes, we assess Triraksha with the UDS framework. The scheme requires software modifications only at the client and the middlebox end. It does not require server side modifications and supports *Server compatibility*. The number of TLS handshakes between a client and a server in Triraksha is one handshake (in Triraksha, the TLS handshake is simply extended to leak TLS secrets to the middlebox) which is the same as in a regular end-to-end TLS connection. It hence complies with *No significant latency*. Triraksha allows for TLS negotiation only between the client and the server.

The client further authenticates the server (during TLS negotiation by checking the server certificate) and middlebox (by using SSH with the rsync connections). It hence complies with *Server authentication*, *Middlebox recognition* and *TLS protocol negotiations*. Triraksha provides fine-grained security as it lets the middlebox access to plain text data only for the connections in which the TLS session secrets are leaked to the middlebox. It complies with *Connection specific interception*. Triraksha does not comply with *Minimal read disclosure* as the Middlebox has access to the plain text for connections that the clients leaks TLS secrets. It complies with *Minimal write access* as the MAC keys are not shared with the middlebox and no modifications can be made to the data sent between the client and the server. It relies on existing and familiar browser mechanisms to display server's certificate and warning signs for insecure connections. Existing schemes like DANE, HSTS etc. that are implemented by an application can also be used in Triraksha. The middlebox is preconfigured to act as a network intermediate (gateway) for a client and hence partially complies with *Middlebox discovery visibility* and *Infrequent errors or open fails*. When the user switches to a new network, the middlebox would no longer be a gateway in the new network (by default) and hence supports *Middlebox persistence visibility*.

4.4.2 Experimental setup

We setup a test environment to evaluate functionality and performance for Triraksha based on the implementation described in Section 4.3. We test Triraksha by making web requests from our university network under two scenarios:

1. HTTP requests to local servers, and
2. HTTP requests to remote servers.

A local server is a server in our local network using the OpenSSL s_server program

[11] while remote web servers represent real world web sites. We use the Chromium and Curl to make single/multiple HTTP web requests to remote and local servers. Chromium uses CCA9 also referenced as ECDHE_RSA_WITH_POLYCHACHA20_SHA or a variation of this suite as the default cipher for its connections. This cipher suite is not recognized by the Wireshark TLS dissector and hence we blacklist the ciphers: CCAX (where X is a numerical variation). All other cipher suites offered by Chromium are acceptable. With Curl, we simply use the command line argument CIPHERS: HIGH to set the cipher suite for the connection. Unless specified, experiments run in the environments consist of a 100 runs for which we report the mean or cumulative result. During the run of these experiments, we do not record the deviation for the average (the numbers for the deviation for the average recorded is minimal). While the environment was setup to make web request automatically to a large number of websites, we perform the experiments manually by making web requests randomly to URLs from the list of Alexa's top 100 websites. Further details on this are provided in the subsequent sections. The benchmarks provided in the performance evaluation are specific to our proof of concept and not to the Triraksha design. The numbers may possibly improve with some other implementation.

4.4.3 Functionality evaluation

To evaluate the functionality for Triraksha, we address the following questions:

1. Can the Triraksha middlebox perform packet inspection correctly?

2. Is Triraksha robust?

We answer these questions by first describing the methodology for the functionality evaluation.

Methodology: We picked websites randomly from the list of Alexa's top 100 websites. We download files of varying sizes (a few Kb upto 10 Mb) using Curl when requests were made to a local server. When connecting to remote servers with Curl, only the headers and index file were downloaded. For Chromium, we simply open the website and let the browser load all objects for the main page of the website. We run experiments manually with Curl and Chromium on approximately 20-30 domains. To test packet inspection, we set the ground truth as plain text decrypted by Wireshark on the client computer under two models. Under model 1, our goal was to check if the hash of the decrypted content for a middlebox and client under Triraksha matches with the hash of the decrypted content for a client in a regular TLS connection (ground truth). Under model 2, our goal is to test if the decrypted content match for the client and the middlebox when both are used with Triraksha.

Model 1: Under model 1, the experiment is run twice for a single web request. For the first experiment, the scheme is a regular end-to-end TLS connection. We enable decryption with Wireshark on the client end and print out all decrypted content for TLS application data packets in hex values to a file. We calculate a hash of the file and store it. In the second experiment, we then perform the same web request under Triraksha and print the decrypted content on the middlebox (as hex values). A hash of the file was calculated and stored.

The hash was done using the Linux sha1sum command and a comparison was made between the hash of the client decrypted content and the hash of the middlebox decrypted content. While the comparison process could be automated to test for a larger number of web requests, we do it manually for two reasons: (1) we are interested in comparing the decrypted content only for a single web request at a time. By making multiple web requests, decrypted content for those requests would be printed to the same file. The only way to cleanly print out the decrypted content

for a single web request is to manually re-initate the capture everytime a web request is made. (2) Making manual web request helped us perform better debugging. It is easier to understand the Wireshark log for a single web request compared to simultaneous multiple requests. Cases in which a match of the hashes does not occur, our script would further run to verify the discrepancy by printing out the non matching decrypted content. We then followed up with manual debugging of non matching decrypted content with the Wireshark SSL debug file and network log.

Model 2: Under model 2, we run the experiment once for a single web request under Triraksha. We store the decrypted content at the client and the middlebox end. We perform a hash of the file and log it. Similar to model 1, we do the experiments manually for websites chosen randomly from Alexa's top 100 list.

Results summary: The hash of the decrypted content matched for the client and the middlebox under both models for Curl requests. We encountered a known Wireshark bug and had to analyze the network logs to understand the exact sequence of events. After applying a manual solution for the Wireshark bug, the hash for the decrypted content matched on both the client and the middlebox end.

Discussion: We discuss the results in detail now. Initially, under model 1, the hash of the decrypted content did not match in any web request made by Curl or Chromium to local or remote servers. The file size of the decrypted content on the client was more than that on the middlebox which implied that the middlebox did not perform decryption for all TLS application data packets. Our debugging script pointed us to the matching and non matching content with the packet number (Wireshark frame number). For Curl requests, we observed that the GET request on the client and the middlebox were decrypted correctly, however, the middlebox would not decrypt the

index file (response text of the server).

On manually debugging the capture file and understanding the Wireshark SSL debug log, we find the reason for this is due to various retransmissions for TCP/TLS packets. In every web request, we had a case of a retransmission for the GET request and the Server ChangeCipherSpec message. We come across bug 9461 [21] in the Wireshark bug forum which states that Wireshark fails to decrypt TLS packets when TCP packets fall out of order. In Wireshark, by default the TCP dissector will hand retransmissions to the subdissector (in this case the SSL dissector) which means that the subdissector sees the retransmitted data twice. This corrupts the state of the SSL dissector and is hence is unable to decrypt the TLS application data packets. The client under a regular TLS connection would decrypt the TLS application data packets and see the index file of the website as usually there are no retransmitted packets in a regular TLS connection. Compared to the previous state, the middlebox in Triraksha drops packets if a firewall rule is not present and hence experiences retransmissions. As a result, Wireshark on the middlebox in Triraksha does not decrypt the TLS application data packets containing the index file. A solution to this is to simply set to ignore the retransmitted packets (done manually by toggling preference for the packet in the GUI) after which the SSL dissector resets its state and is able to decrypt packets correctly. The hash of the decrypted content on the client under a regular end-to-end TLS connection and the hash of the decrypted content on the middlebox and client under Triraksha matched for all cases when we manually toggled to ignore retransmitted packets.

Under model 2, our goal was to test if the decrypted content match for the client and the middlebox when both are used with Triraksha. We follow the same routine as described in the previous model, and additionally we manually set to ignore the

retransmitted packets. We then match the hash of the decrypted content of the client and the middlebox. The hash of the decrypted content matches in all cases with Curl. We got a successful match for the hash of the decrypted content for all experiments with web requests downloading files of various size from local servers. The experiments conclude that our script to add firewall rules and leak TLS secrets to the middlebox work correctly in the overall implementation. In some cases, experiments for single web requests made with Chromium did not match. Unlike Curl, the TCP packets were out of order in many cases with Chromium. Reassembly of packets did not occur on Wireshark properly and hence led to irregular decryption. In some cases, we experienced retransmission of packets leading to new fragment overlapping an old data fragment (TCP fragmentation). After manually ignoring the respective retransmitted packets and triggering decryption again, the hash of the decrypted content matched correctly.

It is possible to export HTTP objects from the captured file in Wireshark. However, this step requires to stop the capture on the middlebox/client. Also, not all objects can be exported via the GUI as the filenames for the exported data does not comply with the OS filename policy. A script was run to find the hash for all exported objects in a directory for both the client and the proxy. We observed that the hash for all media objects were the same.

We also note that in some cases, malformed packet were logged on the middlebox but not on client and vice versa. We also experienced cases in which there were SSL timeouts/handshake failures and there was incomplete capture of handshake data on the client/middlebox. A SSL handshake timeout may occur regardless of using Triraksha or a regular end-to-end TLS connection. We ignore the experiments for these cases in our evaluation as they are irregular cases (we experienced these cases

for a regular end-to-end TLS connection as well).

Robustness for Triraksha To understand robustness for Triraksha, we test Triraksha with different test cases. We discuss the test cases briefly.

Test Case 1: A client leaking TLS secrets to the middlebox is always able to complete the GET request. The response of the GET request from the server is the same as under a regular TLS connection. This test case checks if the firewall rules are correctly put in order and do not block TLS application data packets for a connection after the client leaks the TLS secrets.

Test Case 2: Functionality check by making two simultaneous connections to the same domain and see if they are recognized differently by the middlebox and client. This test case checks if a client can access resources from a domain (web server) over a connection in which he leaks the TLS secrets while he opens another connection (this time does not leaking the TLS secrets) and attempts to access resources from the same domain (web server).

Our implementation handles this test case by tracking TCP connections. We add a firewall rule for a source IP address, destination IP address, source port and destination port and remove the rule when a TCP FIN/RST packet is detected for this connection. A separate TLS connection to the same IP address will always be opened over a different source port (browser/OS policy). The client cannot use the same source port and IP address pair to open another TLS connection as a single TCP connection encapsulates only a single TLS connection. We confirm that a new connection is created with the same (ip.src, ip.dst, tcp.src_port, tcp.dst_port) after an

authorized one was closed (either by timeout or by remote server closing the connection, or by client closing the connection).

Test Case 3: The client always sends the TLS secrets to the middlebox when the application making the web requests is exported with the SSLKEYLOGFILE. To test this case, we run a script to make multiple Curl requests to Alexa's top 5000 websites. After the script is run, we check the hash of the generated TLS secrets on the client computer and the hash of the TLS secrets received by the middlebox.

Test Case 4: An application not leaking TLS secrets cannot communicate TLS application data packets with the server and vice versa. This rule checks if the TLS application data packets are dropped correctly on the respective network interface.

Test Case 5: We send incorrect and correct TLS secrets to the middlebox and confirm that the decrypted text is respectively garbage/correct in this experiment.

Test Case 6: We verify that a long processing time at the middlebox doesn't interfere with the TLS handshake (because of retransmissions, timeouts, several ClientHello sent by browsers at once, etc.).

We evaluate Triraksha on these test cases by running experiments manually. We made web requests to domains from Alexa's top 100 list. Initially, some of the test cases failed (due to retransmission of packets, minor errors in implementation etc.) but were fixed over the time period of this research. We would like to note that the test cases were developed to stress test the proof of concept. They were written to test each components of the Triraksha implementation and if the components interact

with each other the way they are supposed to. Triraksha successfully passes on all of these test cases.

4.4.4 Performance evaluation

In this section, we are interested in finding the overhead incurred by Triraksha compared to a regular end-to-end TLS connection. We start by discussing detailed micro benchmarks for the client and the middlebox with web requests made with Curl. The timing information was collected by leveraging Curl's built-in timing APIs.

Client Performance: *How long does it take to generate keys and send it to the middlebox during the handshake?* Table 2 provides some insight into the logistics for the time taken to generate TLS secrets based on our implementation. We use six popular websites for our experiments: Youtube, CNN, Twitter, Facebook, YTS and New York Times. The websites together all have different types of media content. On an average, Wireshark on the client side takes 344 μs to read a SSLKEYLOGFILE, generate TLS secrets and write them to a file. We measured the time interval to run the respective functions collectively by using APIs from the glibc library for this statistic.

Curl's appconnect metric gives the time, in seconds, it takes from the start until the SSL/TCP/ connect/handshake to the remote host is completed. With the Triraksha handshake, the handshake includes sending the TLS secrets to the middlebox. On an average, sending the TLS secrets using rsync for a single Curl request takes 280 ms. Table 3 contains statistics for the rync transfers. Table 4 represents the statistics for Curl's appconnect metric. The metrics in Table 4 were established over a period of 5 runs. From the table, we observe that the Triraksha handshake has little or no

[1]Definitions for Curl's APIs are in the Appendix.

overhead compared to a regular TLS connection. This is expected as we do not make any changes to the TLS handshake itself. The TLS handshake protocol for Triraksha and a regular TLS connection is the same and the key generation process and transfer of TLS secrets do not affect the handshake. There is only network latency observed in Curl's appconnect metric for Triraksha and a regular TLS connection. The overhead incurred in the Triraksha handshake is attributed to sending the TLS secrets to the middlebox.

Website	Time taken (μs)
Youtube	426
Twitter	381
CNN	392
Facebook	241
YTS	332
New York Times	293

Table 2: The time taken to read the SSLKEYLOGFILE and generate TLS secrets.

Website	Time taken (ms)
Youtube	283
Twitter	291
CNN	286
Facebook	294
YTS	286
New York Times	288

Table 3: The time taken to send TLS secrets using rsync.

Summary: There is little overhead for the Triraksha handshake when compared to a regular end-to-end TLS handshake. The additional overhead in Triraksha handshake is from the latency of sending the TLS secrets from the client to the middlebox.

Middlebox performance: When it comes to evaluating the performance for the middlebox, we are interested in addressing two questions: *How long does it take to*

Website	TLS (ms)	Triraksha appconnect (ms)	Total time Triraksha (ms)
Youtube	105	136	419
Twitter	179	176	459
CNN	172	166	452
Facebook	215	184	478
YTS	31	37	323
New York Times	102	158	446

Table 4: The time taken for Curl's time_appconnect (handshake time) and the total Triraksha handshake time.

create a dummy SSLKEYLOGFILE and add rules to the firewall once the middlebox receives the TLS secrets? and *What is the total time taken to complete a single Curl request with Triraksha?* Table 5 gives details for the time taken to execute the scripts and add rules to the firewall. The time taken by IP sets to add firewall rules is minimal (an average latency of 12 ms). We do not measure the time taken for packets to be matched against the respective firewall rules.

Website	Time taken (ms)
Youtube	12
Twitter	14
CNN	13
Facebook	12
YTS	13
New York Times	12

Table 5: The time taken to create a dummy SSLKEYLOGFILE and add rules to the firewall.

We now measure and show the time taken to complete the entire Curl request. The time taken to complete the entire Curl request consists of the time taken to do a DNS lookup, time for the handshake to complete, sending the TLS secrets to the middlebox, adding a firewall rule and the time taken to transfer the server response for the website. Table 6 shows detailed statistics for a website. The major difference between Triraksha and a regular end-to-end TLS connection lies in Curl's

'time_starttransfer'. time_starttransfer is the time, in seconds, from the start until the first byte is transferred. This includes the time taken for the handshake, redirects, the DNS name lookup and the time the server needed to calculate the result. Table 8 provides a summarized version of the total time taken for the six websites used in previous experiments.

From the tables, we observe that the time taken to start transfer for application data packets initially is significant. On average, an overhead of 1.2 seconds occurs across the websites. On manual analysis (for each website) of the packet capture running with Wireshark, we find re-transmitted packets at the TCP and TLS layer. Specifically, the GET request from the client is retransmitted multiple times (3-10 times) and the ChangeCipherSpec from the server is logged as a TCP spurious retransmission in all instances. TCP spurious retransmission indicates that the sender sent a retransmission for data that was already acknowledged by the receiver and for some reason, the sender interpreted that a packet was lost, so it sends it again. Further, the retransmission time roughly doubles for every retransmitted packet (adaptive TCP retransmission policy). The time difference in establishing a TLS connection with Triraksha and a regular end-to-end TLS connection is attributed to re-transmissions of packets. We should note that time_starttransfer is the time taken to first byte (TTFB) for the TLS application data packet. If the connection were to remain open and a resource was downloaded from the server using the same TLS connection then the difference between the two schemes would be negligible. Table 10 show statistics

Property	End-to-End TLS	Triraksha
time_starttransfer	0.503 seconds	1.839 seconds
time_total	0.523 seconds	1.942 seconds
size_header (at client)	763 bytes	763 bytes
size_download (at client)	34548 bytes	34548 bytes

Table 6: The time and size for a Curl request for yts.ag.

Property	End-to-End TLS	Triraksha
time_starttransfer	0.519 seconds	1.32 seconds
time_total	0.664 seconds	1.877 seconds
size_header (at client)	1364 bytes	1364 bytes
size_download (at client)	309066 bytes	309066 bytes

Table 7: The time and size for a Curl request for Twitter.com.

Website	End-to-End TLS (s)	Triraksha (s)
Youtube	0.18	1.08
Twitter	0.66	1.87
CNN	0.29	1.12
Facebook	0.46	1.22
YTS	0.523	1.94
New York Times	0.17	1.01

Table 8: The total time taken for a Curl request compared to a regular end-to-end TLS connection.

Property	End-to-End TLS	Triraksha
time_starttransfer	0.22 seconds	0.9 seconds
time_total	0.29 seconds	0.906 seconds
size_header (at client)	331 bytes	331 bytes
size_download (at client)	66459 bytes	66459 bytes

Table 9: The time and size for downloading a Thunderbird addon: cherami.xpi.

Property	End-to-End TLS	Triraksha
time_starttransfer	0.16 seconds	0.19 seconds
time_total	39.03 seconds	39.12 seconds
size_header (at client)	394 bytes	394 bytes
size_download (at client)	33852374 bytes	33852374 bytes

Table 10: The time and size for downloading a repo tarball from OpenSSL.

for the timing difference in downloading a larger file. The time difference between Triraksha and a regular TLS connection for downloading a 33MB file is 0.09 seconds. The difference is very small and attributed to network latency.

Summary: There is an overhead initially only for establishing the TLS connection. Once the TLS connection is established and firewall rules are added, other resources from the server take the same time to be downloaded as a regular end-to-end TLS connection.

Performance overhead with browser requests: Triraksha can be adopted with popular desktop web browsers like Chromium, Firefox etc. To understand how benchmarks with a Curl request transform with real world browsers, we evaluate Triraksha with web requests made from the Chromium browser. We choose websites from the list of Alexa's top 100 websites. We record all information from the network log using Chromium's developer tool and the decrypted plain text by Wireshark. For each single web request, we record logistics for the objects downloaded and uploaded. The information was stored as a HTTP Archive format file (HAR) and was analyzed with a HAR viewer. Tables 11 and 12 show performance benchmarks of two popular websites for Triraksha compared to a regular end-to-end TLS connection.

Property	End-to-End TLS	Triraksha
No. of requests	26	51
Onload time	1388 ms	5832 ms
SSL time	365 ms	412 ms
Uploaded data	15273 bytes	29365 bytes
Downloaded data	796098 bytes	3456866 bytes

Table 11: The time and size for a Chromium request (page global data) for Amazon.ca.

[1]Definitions for parameters involved with browser experiments are in the Appendix.

Property	End-to-End TLS	Triraksha
No. of requests	36	43
Onload time	2401 ms	10220 ms
SSL time	250 ms	581 ms
Uploaded data	24026 bytes	27009 bytes
Downloaded data	1310201 bytes	1315904 bytes

Table 12: The time and size for a Chromium request (page global data) for eztv.ch.

The difference between the schemes mainly lie in the uploaded and downloaded data. Each retransmitted packet on Chromium between the client and the server increments the stats reflecting the difference in the downloaded and uploaded data from the HAR file. Overall, the browsing experience for both the schemes is very similar.

Data volume overhead: The data volume overhead for a middlebox in Triraksha is from the TLS secrets received by client. The memory consumption on the middlebox would increase with the number of TLS connections. For each TLS connection, the secrets consists of 56-64 bytes. The TLS secrets are send in a separate connection and take only a few bytes of disk space. There is no size difference between the Triraksha handshake and a regular TLS handshake; however, data volume overhead exists from the retransmission of packets (cannot be quantified due to its variable nature).

Data throughput: The data throughput for our scheme ideally should be the same as for a regular TLS connection. However, the number of connections is limited by the resources on the middlebox. With our current implementation setup, the modified version of Wireshark can decrypt upto approximately 5000 Curl requests around which and after Wireshark experiences memory crashes. We made Curl requests to a list of Alexa's top 10000 websites. After a number of runs for this experiment, we

[1]Machine configuration for the proof of concept is in the Appendix.

observed that Wireshark is stable upto 5000 connections.

CPU overhead: Triraksha performs worse compared to Split TLS and other schemes when it comes to CPU overhead. The scripts run to achieve the intermediate steps in Triraksha are event and I/O driven. The resources used by the scripts are minimal, however, the main source of CPU overhead is attributed to the resources consumed by Wireshark. Wireshark is used for the generation of TLS secrets on the client end and as a decryption engine at the middlebox end. Wireshark uses significant memory as it frequently updates a GUI component, captures packets and dissects them. Further, Wireshark dissectors do not run in a multi-threaded environment. Using Wireshark on the client is a tradeoff between deployability and resource consumption. Modifying the browser or underlying SSL library to leak the TLS secrets would result in the enterprise to adopt only that version of the browser/SSL library. Patches for every update to browsers/SSL library would have to be made and this would increase the load on the administrators of the network.

On the middlebox end, Wireshark is used to provide access to the plain text. The overhead of resource consumption by Wireshark for decrypting packets would increase linearly with the number of connections made by the client and the size of the TLS packets. In a production environment where resources for the middleboxes are higher, we do not recommend to run Wireshark to capture data for long periods as Wireshark may itself suffer from memory leaks or other vulnerabilities.

Triraksha design does not implicate high CPU overhead. In practice, any other software which uses less CPU resources can be used for generation of TLS secrets/as a decryption engine. Wireshark provides a neat way to see the network log in a GUI (which helped us perform debugging and evaluation), but in practice, the GUI

is not really required for the functionality. We alternatively use the command line version TShark which can also achieve the same functionality as Wireshark and uses significantly lesser resources. TShark does not fork a separate child process to handle capture of packets like in Wireshark and does not have a GUI component. We apply filters to capture only TLS packets and discard the rest. This reduces the size of the capture file.

Chapter 5

Discussion and extensions to Triraksha

In this chapter, we discuss a possible extension for Triraksha to perform MAC verification for the decrypted payload in a connection and how Triraksha handles Authenticated Encryption with Associated Data ciphers (AEAD).

5.1 TLS 1.3 and AEAD ciphers

5.1.1 TLS 1.3

TLS has a long history of vulnerabilities based on implementation and cryptographic security. The IETF (Internet Engineering Task Force) has been developing TLS 1.3 standard which would be the defacto standard for cryptographic protocols in the TLS family. While the changelog extends to significant improvements like improving round trip times (RTT) for protocol messages, removing compression, improving downgrade protection, adding a full handshake signature etc., we are interested in the improved and robust crypto standard provided by TLS 1.3. TLS 1.3 removes

obsolete and insecure standards for cryptographic primitives. In short, all primitives that would lead to a weak TLS configuration are being removed For example, RC4, DES, 3DES, EXPORT-strength ciphers, weak and rarely-used elliptic curves, AES-CBC, MD5, and SHA-1 are no longer part of TLS 1.3. The current list of cipher suites in TLS 1.3 is subject to AEAD ciphers, mainly: AES-GCM, AES-CCM, ARIA-GCM, Camellia-GCM, ChaCha/Poly. Besides this, TLS 1.3 does not support the old static RSA handshake without Diffie Hellman as the static RSA handshake does not support perfect forward secrecy. TLS 1.3 removes explicit nonces and support for re-negotiation.

5.1.2 AEAD ciphers

Given the release updates in the TLS 1.3 draft, Triraksha should support the upcoming TLS protocol. However, Triraksha does not work well with AEAD ciphers. We discuss this and introduce AEAD ciphers here.

Authenticated Encryption with Associated Data (AEAD) is a form of encryption which provides confidentiality, integrity, and authenticity assurances on the plaintext. Authenticated encryption (AE) schemes are typically constructed by combining an encryption scheme and a message authentication code (MAC). Approaches to such schemes involve Encrypt then MAC or MAC then encrypt. AE schemes have evolved to use encryption and MAC under a single interface for use with block ciphers. An example of this approach is GCM (Galois counter mode). For the purpose of this section, we are interested in the AEAD ciphers that encrypt and MAC under a single interface and are supported by TLS 1.3. These ciphers involve the following schemes: CCM, OCB, GCM, EAX etc. OCM is a strong AEAD cipher, but is patented and is legally challenging to use in practice. CCM is a two-pass mode and does not work well for streaming applications like Netflix and Twitch. GCM is a popular cipher and

is well known for its performance and efficiency as it parallelizable. It is commonly used in SSL/TLS. The strength of AE implies IND-CCA2 (in-distinguishability under adaptive chosen ciphertext attack) and NM-CCA2 (non-malleability under adaptive chosen ciphertext attack) security.

We shall now describe the functioning for GCM, why it does not work with Triraksha and how to tweak it such that it can be incorporated in our scheme.

GCM and GMAC: GCM is a mode of operation for symmetric key cryptographic block ciphers to provide data encryption and authentication. Galois Message Authentication Code (GMAC) is an authentication-only variant of the GCM which can be used as an incremental message authentication code. Figure 7 represents the working of GCM.

GCM has two operations [56]: authenticated encryption and authenticated decryption. The authenticated encryption takes the following inputs:

1. A secret key K.
2. An initialization vector IV of length 1 and 2^{64} bits. IV should be unique for each key.
3. Plain text P for which encryption is performed.
4. AD (Additional data) The data is authenticated but not encrypted. Length of AD can be 0 and 2^{64} bits.

There are two outputs:

1. A cipher text C (length is equal to plaintext P).
2. An authentication tag.

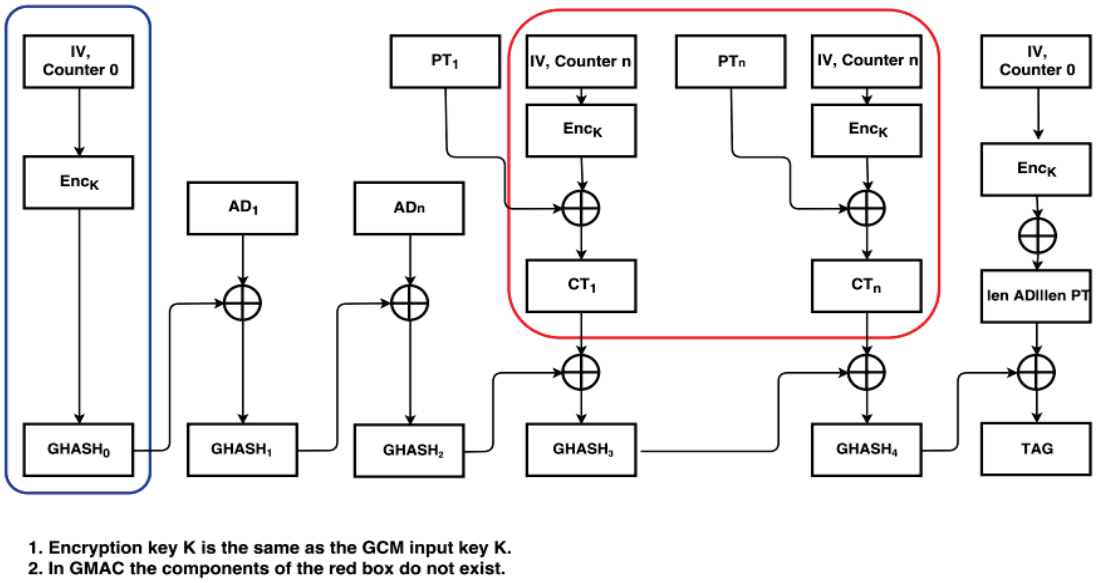

Figure 7: Internal functioning of the GCM encryption operation.

The authenticated decryption operation has five inputs: K, IV , C, AD, and T. It has only a single output which is the plaintext value P or a symbol 'FAIL' that represents that the MAC failed for the input values. The AD is used to protect additional information that needs to be protected, for example, in a network protocol AD can be the IP address and port numbers. When the length of the plain text is zero, GCM acts as a MAC on AD. This mode is called GMAC. The IV acts as a nonce that is to be unique for each invocation of the encryption operation for a fixed key. For example in TLS, the IV is split into two parts: the implicit salt (4 bytes, generated in handshake, not changed in the whole session) and a explicit nonce of 8 bytes, chosen by the sender and carried in each SSL record (usually the TLS sequence number for a record). The combination of the implicit salt and the explicit nonce makes the IV unique. The two main functions used in GCM are the block cipher encryption and multiplication over the field GF(2^{128}).

Issue with GCM and Triraksha: The issue with using GCM in Triraksha with minimal write permissions is that the MAC key is not revealed to the middlebox, only the decryption key; however, in GCM the MAC key (which is GHASH_0 in Figure 7) is derived from the decryption key (which is K). Specifically $\text{GHASH}_0 = \mathsf{Enc}_k(000\ldots0)$. Thus by revealing K, you effectively reveal GHASH_0. There is no security reason that the MAC key needs to computable from the decryption key, however GCM does require that only one key is passed into the encryption/decryption operations. Therefore, we suggest two solutions that would change the internal functioning of GCM in a way that would allow Triraksha, as well as any other application where one might want to selectively disclose sub-keys to allow only reading or only writing to ciphertexts.

1. Consider the input key to GCM. Call it MK, for master key, instead of K in Figure 7. We use MK to derive two independent subkeys using a counter: an encryption key and a MAC key. Specifically: $\text{K} = \mathsf{Enc}_m k(000\ldots0)$ (encryption on counter 0) and $\text{GHASH}_0 = \mathsf{Enc}_m k(000\ldots1)$ (encryption on counter 1). See Figure 8. This comes at the expense of an additional encryption (e.g., AES) operation. We share only the new encryption key with the middlebox.

2. Alternatively to avoid the addition of an extra encryption operation, we could simply swap the roles of K and GHASH_0. The key supplied to the GCM operation could be consider the MAC key and thus GHASH_0, while the value derived from it can be considered the encryption key. Specifically, $K = \mathsf{Enc}_{\text{GHASH}_0}(000\ldots0)$. We share only the new encryption key with the middlebox.

GCM is used in two modes: encryption/decryption with authentication and GMAC (only authentication). We argue that both the approaches do not break GCM when it is used to do encryption/decryption operations or when used in GMAC mode. We do

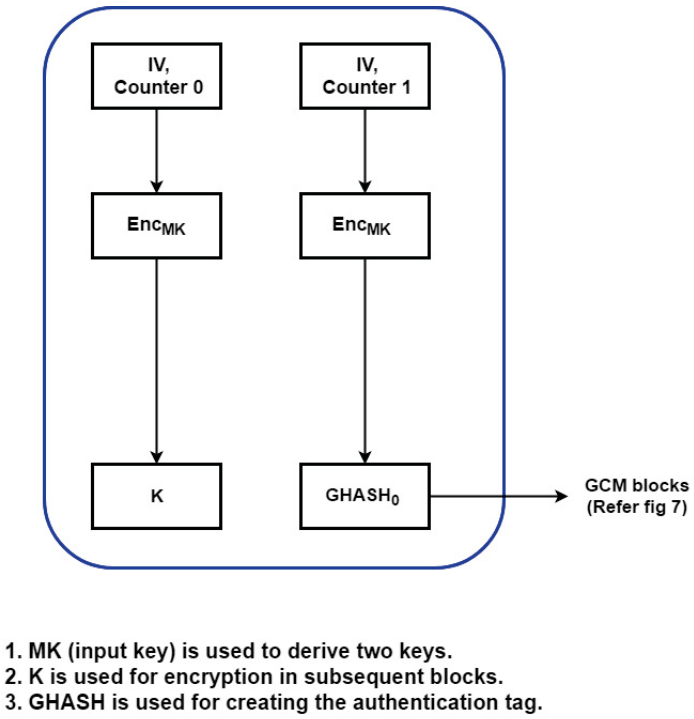

Figure 8: Deriving two keys from MK in GCM.

not make any changes to the way GCM performs encryption or MAC authentication. The only changes we make to GCM involve deriving values for the encryption key and the MAC key (which would be used for the block cipher encryption of the plaintext) or swapping the value of the MAC key and the encryption key after the MAC key is generated.

We also note that in this case Triraksha would not be server compatible as the server would also have to be modified to use the same TLS library as used by the client. The TLS library would have to be modified to support both the approaches.

5.2 Verification of TLS secrets and integrity of data using MAC

Our current threat model allows the middlebox to trust the TLS secrets provided by the client. Consider a model where we have a 'malicious' user who tries to bypass the Triraksha protocol. The malicious user can simply send the middlebox bad/wrong TLS secrets. The client can communicate messages with the server and the middlebox would simply decrypt the traffic with the wrong TLS secrets resulting in garbage decryption. We extend the Triraksha threat model to handle a malicious user. Under this model, we let the middlebox perform MAC verification for the decrypted text. The middlebox can then detect if the TLS secrets provided by the client are correct. The extended model only helps in detection for integrity of data. A middlebox operator would be aware if MAC verifications fails for the decrypted plaintext. We do not support prevention at the middlebox end; that is, even on MAC verification failure for the TLS application data packets, we let TLS application data packets for that connection to pass through (MAC verification is done only after packets pass through the middlebox).

The extended model lets the middlebox trust the client for a grace period until it receives the master secret for the TLS connections recorded by Wireshark. Wireshark can use the master secret to verify integrity of the symmetric encryption keys and the data by performing MAC verification. The MAC key derived from the master secret will authenticate the payload of the TLS application data packets. The process described above uses the GUI functionality provided by Wireshark to reload the SSLKEYLOGFILE. This runs the TLS dissector code again decrypting the captured packets based on the master secret provided in the new SSLKEYLOGFILE.

Under our threat model, the middlebox can introspect data but is not capable

of modifying it. We provide the master secret to the middlebox only when we know that a TLS connection has ended or after it is re-negotiated. The latter approach, which is to induce re-negotiations allowing us to set a grace time period after which a connection would be renegotiated. By inducing forced re-negotiations for TLS connections after a given time period we can give the middlebox the master secret. The new negotiated handshake would have a different master secret for the client and the server. The middlebox can no longer have access to the correct MAC key for the current TLS connection and can perform MAC verification for the previously recorded data with the old master secret provided by the client. Re-negotiation in TLS usually happens in the client when:

1. The server requires a re-negotiation, typically because the client tries to access a resource which requires a client certificate which the previous handshake did not include; or

2. A re-negotiation is done for security reasons after some time or number of bytes transferred. In OpenSSL, this can be tuned with BIO_set_ssl_re-negotiate_bytes and BIO_set_ssl_re-negotiate_timeout; or

3. The 64-bit TLS sequence number overflows.

Since we do not make changes at the server end, we leave it to the application developer using Triraksha to recompile the underlying TLS library/application to set a re-negotiation time out period. With TLS 1.3, re-negotiation for TLS sessions are no longer supported. In this case, we only provide the master secret to the middlebox when the TLS connection has ended. In our implementation, we track the state of a TLS connection by detecting TCP FIN packets. Whenever the client detects a TCP FIN packet, it means the connection has ended and the client can share the

master secret with the middlebox. The middlebox would then use the master secret to perform MAC verification of previously recorded data.

5.3 Extending Triraksha use cases and a note on usability

In Triraksha, the client and the server continue to main integrity for messages sent between them. The Triraksha middlebox is limited in its use cases. The middlebox cannot perform functionality like compression and cache which requires modification of the traffic sent between the client and the server. The Triraksha protocol can be extended such that we also share the MAC key with the middlebox. This enables the Triraksha middlebox to be used in other scenarios as well. We suggest to make this decision only when the middlebox is trusted and perceptible by the client and the server. A consequence to this decision would be that the middlebox can now modify the traffic in a connection and the client and server would loose integrity for the messages between them.

Triraksha is presented as a scheme that provides client side consent to the middlebox. In practice, Triraksha can be modified to provide server side consent to the middlebox. While we do not list the specifics, the concept remains similar. The server would leak TLS secrets to the middlebox instead of the client. This enables the Triraksha middlebox to be used as a reverse proxy or a load balancer.

Scalability compared to Split TLS and possible improvements: The Triraksha middlebox ideally can handle equal or more number of clients compared to Split TLS. Relative to Split TLS, for each connection in Split TLS, the middlebox needs to open 2 sockets for connections between the client and the server and the

client and the middlebox. In Triraksha, the middlebox needs to open 2 sockets: one for handling the TLS secrets from the client and one for forwarding packets between the client and the server.

In some RFCs, the TLS session encryption key/TLS secrets (encrypted with the middlebox's public key assuming RSA encryption key in middlebox's cert) is included in an ignorable header field in the TLS packet header itself. The header is added after a certain point in the handshake or after knowing the TLS encryption key and is packed into the TLS packet and parsed by the middlebox. The packet is parsed before it reaches the server. With Triraksha, this kind of a scheme design (which would involve opening only a single socket respective to a connection on the middlebox) can be explored to improve scalability.

Usability configuration for end user and middlebox: In Triraksha, usability for the end user and the middlebox administrator can be achieved by setting policies. The objective for setting such policies is to communicate to the user 'who can do what' in a simple and scalable manner. The policy communicates if a middlebox can read/write the user's data?

At the client end, for each website/group of websites, an user can set a policy such that the middlebox would be allowed:

1. No read permission: The client does not leak the TLS secrets in this case.
2. Only read permission: The client leaks the TLS secrets to the middlebox.
3. Read and write permissions: The client includes the MAC key with the TLS secrets used in Triraksha and leaks the TLS secrets to the middlebox.

Similarly the middlebox can be configured to have a central policy to allow/block websites. Further, a policy can be configured on the middlebox for a website/group of websites such that:

1. The middlebox does not intercept the connection. The middlebox would not block TLS application data packets for this website and does not require the TLS secrets from the client.

2. The middlebox requires to inspect packet payload. The middlebox follows the Triraksha protocol and requires the client to leak the TLS secrets.

3. The middlebox requires read and write privileges over the traffic. The client would have to include the MAC key with the TLS secrets and leak it to the middlebox.

A note on the browser security indicator: HTTPS over the years has matured and integrated with popular browsers visualizing to the user if the connection is trusted or not. Browsers are one of the most popular applications used by a user for making TLS connections to a server. A standard HTTP with SSL connection in a browser is graphically shown to the user with a lock icon. The intended goal of using the lock icon is to provide the user with the identity of page origin and indicate to the user that page contents are not viewed or modified by a third party. However, displaying the origin ID in the padlock is not always helpful as origin ID can easily be manipulative and provide the user with similar looking origins. Using the padlock icon tells the user that all the elements are fetched from a trusted server. Trust is reify by performing security validations like if the HTTPS certificate is issued by a CA that is trusted by the browser, if the HTTPS certificate is valid and if the common name in the certificate matches the domain name in the URL. Most browsers recognize EV (extended validation) certificates which are designed for large scale banks and

e-commerce websites. EV certificates provide the strongest encryption level available and enable the organization behind a website to present its own verified identity to website visitors and help block semantic attacks (phishing etc). A green colored address bar is shown in browsers for the usage of EV certificates.

The visual cues for SSL (lock icon and green colored bar) sometimes do not not reflect their intended purpose and fail to show the user, the trust of the connection. There are number of attacks [76] identified by researchers on a SSL connection that can trick the user to have a false sense of security while having the padlock icon shown to them. SSL strip is one such attack in which an attacker can redirect the server address to the attacker's page where he can use similar looking lock icons on the website instead of the address bar and trick a gullible user to enter his credentials. There can be semantic attacks on certificates where the attacker can buy a certificate for a domain looking similar to the legitimate domain name. The user may be tricked into thinking he is visiting a legitimate website. Sometimes when certificates are invalid and there are pop up warnings in the browser, user often ignore these warnings thinking of it as a misconfiguration rather than an attack. It takes as much as 4-6 mouse clicks to accept invalid certificates in some of the browsers. In the case of a server sending mixed content (HTTP and HTTPS), the padlock icon is modified to show a '!' along with the lock icon for Firefox. Safari does not detect mixed content and no warnings or prompts are issued to the user. In a MITM attack the user can be easily tricked into accepting a certificate from the attacker. The user would still be shown a padlock icon and confidentiality is no longer maintained. Joshual et al. [76] in his paper does an empirical study of SSL warning effectiveness. They recommend to block users from making unsafe connections rather than displaying warnings in benign situations. Schechter et al. [68] perform an evaluation of website authentication and conclude that users enter credentials even when the passive HTTPS indicators are

absent. They also conclude that authenticated website lookalike images make users disregard other security indicators.

Designing an interface for browsers: Browsers are one of the most popular TLS applications. There is a need for browsers to have some kind of a security indicator to visualize to the user the existence of a middlebox and to indicate trust for the connections. Reusing the padlock seems to further mislead users and fails to indicate if the connection is 'secure'. In the proposal: any node refusal, the author suggests to use a double padlock icon, each lock displaying a secure connection for the server and a middlebox. We leave designing a satisfactory 'connection security indicator' for browser as future work.

Security indicator for Triraksha: Other common user applications which make TLS connections on a general note do not have a connection security indicator and do not display trust for the connection to the user (example: Spotify, Skype etc.). With Triraksha, we envision a 'VPN style indicator' where the OS is aware if the TLS connection is being proxied. A simple interface (example: button, notification) can indicate if the client is sending TLS secrets for a connection. This extends visibility for the connection security indicator to all applications and not just the browsers. An OS level security indicator can support application layer gateways as well as proxies (the latter approach of using a padlock icon works only for middleboxes that act as proxies). We leave designing a satisfactory 'connection security indicator' for Triraksha as future work.

Chapter 6

Conclusion

Middlebox are increasingly being deployed by organizations and provide a wide array of valuable in-network services. Simultaneously, end-to-end encryption is adopted increasingly across the Internet. Middleboxes do not work well with end-to-end encrypted sessions. The naive and popular solution Split TLS breaks end-to-end encryption and is not secure. While, there are other models that incorporate middleboxes with end-to-end encryption, they are not adopted as they are not easy to deploy in the real world. These schemes have varying threat models and their end goals are not aligned with the deployment needs of users and service providers.

In this work, we introduce a benchmark based on the UDS framework that encompasses the threat models, entities and properties of the schemes and provides a clear insight into the challenges and needs of using middleboxes with end-to-end encrypted sessions. We introduce our scheme Triraksha which provides a packet inspection service for TLS connections. Triraksha design and implementation highlights challenges for a real world deployable scheme. We hope that researchers will continue to explore the standing problem of enabling end-to-end encryption with middleboxes for future versions of the TLS protocol.

Bibliography

[1] Any node refusal. https://github.com/bizzbyster/TrustedProxy/wiki/Any-Node-Refusal.

[2] Ars technica. ATT fined 25 million after call center employees stole customers data. http://arstechnica.com/techpolicy/2015/04/att-fined-25-million-after-call-centeremployees-stole-customers-data/. Article published on 04/2015.

[3] Delegation Oriented Architecture (DOA). https://suricata-ids.org/.

[4] IP sets. http://ipset.netfilter.org/.

[5] iptables. http://ipset.netfilter.org/iptables.man.html.

[6] List of user stories for HTTP proxies. https://github.com/http2/http2-spec/wiki/Proxy-User-Stories.

[7] mitmproxy. https://mitmproxy.org/.

[8] Named Data Networking. https://named-data.net/project/.

[9] NDPI. http://www.ntop.org/wp-content/uploads/2013/12/nDPI_QuickStartGuide.pdf.

[10] nfqueue. http://www.netfilter.org/projects/libnetfilter_queue/.

[11] Openssl server. https://wiki.openssl.org/index.php/Manual:S_server(1).

[12] OSI model. http://www.itu.int/rec/T-REC-X/en.

[13] Qosmos deep packet inspection and metadata engine. http://www.qosmos.com/products/deep-packet-inspection-engine/.

[14] rsync. https://linux.die.net/man/1/rsync.

[15] Scapy. http://www.secdev.org/projects/scapy/.

[16] Snort. https://www.snort.org/.

[17] Suricata. https://suricata-ids.org/.

[18] TCPcrypt. http://www.tcpcrypt.org/.

[19] TLS mailing list, industry concerns. https://www.ietf.org/mail-archive/web/tls/current/msg21278.html.

[20] Wireshark. https://www.wireshark.org/.

[21] Wireshark bug 9461. https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=9461.

[22] Network traffic statistics for encryption. https://www.sandvine.com/trends/encryption.html, 2015.

[23] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel: Google's data compression proxy for the mobile web. In *NSDI*, volume 15, pages 367–380, 2015.

[24] Palo Alto Networks. Statistics for HTTPS. https://www.paloaltonetworks.com/.

[25] Sheehy Andrew. The structure of the internet. http://www.forbes.com/sites/andrewsheehy/2015/09/21/the-structure-of-the-internet-is-going-to-change-radically-in-the/-next-ten-years/6cca8c6f1372. News article 09/2015.

[26] Michael Backes, Rainer W Gerling, Sebastian Gerling, Stefan Nürnberger, Dominique Schröder, and Mark Simkin. Webtrust–a comprehensive authenticity and integrity framework for HTTP. In *International Conference on Applied Cryptography and Network Security*, pages 401–418. Springer, 2014.

[27] K Bhargavan, A Delignat-Lavaud, A Pironti, A Langley, and M Ray. Transport Layer Security (TLS) session hash and extended master secret extension. Technical report, 2015.

[28] Benn Sapin Bollay. Strong SSL proxy authentication with forced SSL renegotiation against a target server, August 4 2015. US Patent 9,100,370.

[29] Benn Sapin Bollay and Erick Nils Hammersmark. Aggressive rehandshakes on unknown session identifiers for split SSL, December 8 2015. US Patent 9,210,131.

[30] Benn Sapin Bollay, David Alan Hansen, David Dean Schmitt, and Jonathan Mini Hawthorne. Proxy SSL handoff via mid-stream renegotiation, October 20 2015. US Patent 9,166,955.

[31] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 553–567. IEEE, 2012.

[32] Brian Carpenter and Scott Brim. Middleboxes: Taxonomy and issues. *RFC 3234 IETF draft Category: Informational*, 2002.

[33] Privacy Rights Clearinghouse. A chronology of data breaches, 2009.

[34] CloudFlare. Keyless SSL. `https://www.cloudflare.com/keyless-ssl/`.

[35] Mark Charles Davis, David G Kuehr-McLaren, and Timothy Glenn Shoriak. Extending SSL to a multi-tier environment using delegation of authentication and authority, April 2 2002. US Patent 6,367,009.

[36] Jayson DeMers. How the internet will change? `http://www.forbes.com/sites/jaysondemers/2016/04/18/7-predictions-for-how-the-internet-will-change-over-the-next-15-years/711b2a5a78dc`. News article 04/2016.

[37] Tim Dierks and E Rescorla. The Transport Layer Security (TLS) protocol version 1.2. *IETF, RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685*, 2008.

[38] Tim Dierks and E Rescorla. The Transport Layer Security (TLS) protocol version 1.3. *RFC 6961, IETF*, 2015.

[39] Jeffrey Erman, Alexandre Gerber, Mohammad Hajiaghayi, Dan Pei, Subhabrata Sen, and Oliver Spatscheck. To cache or not to cache: The 3G case. *IEEE Internet Computing*, 15(2):27–34, 2011.

[40] Oprescu et al. A framework for consent and permissions in mediating TLS. *MaRNEW Workshop*, 2015.

[41] Thomas Fossati, Vijay K Gurbani, and Vladimir 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, pages 1–6. ACM, 2015.

[42] Charles E Gero, Jeremy N Shapiro, and Dana J Burd. Terminating SSL connections without locally-accessible private keys, December 14 2012. US Patent App. 13/714,656.

[43] Mohammad A Hoque, Matti Siekkinen, and Jukka K Nurminen. On the energy efficiency of proxy-based traffic shaping for mobile audio streaming. In *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*, pages 891–895. IEEE, 2011.

[44] Jeff Jarmoc and DSCT Unit. SSL/TLS interception proxies and transitive trust. *Black Hat Europe*, 2012.

[45] Suhairi Mohd Jawi, Fakariah Hani Mohd Ali, and Nurul Huda Nik Zulkipli. Nonintrusive SSL/TLS proxy with JSON-based policy. In *Information Science and Applications*, pages 431–438. Springer, 2015.

[46] Tushar Kanekar and Sivaprasad Udupa. Systems and methods for optimizing SSL handshake processing, July 29 2014. US Patent 8,793,486.

[47] Mangesh Kasbekar. HTTPS request enrichment, September 29 2015. US Patent App. 14/868,771.

[48] A Kingsley-Hughes. Gogo in-flight wi-fi serving spoofed SSL certificates. *ZDNet January*, 5, 2015.

[49] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: securely outsourcing middleboxes to the cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 255–273. USENIX Association, 2016.

[50] Peter Lepeska. Trusted proxy and the cost of bits. In *IETF proceedings*, 2014.

[51] Chris Lesniewski-Laas and M Frans Kaashoek. Ssl splitting: Securely serving data from untrusted caches. *Computer Networks*, 48(5):763–779, 2005.

[52] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. When HTTPS meets CDN: A case of authentication in delegated service. In *Security and privacy (sp), 2014 ieee symposium on*, pages 67–82. IEEE, 2014.

[53] S Loreto, J Mattsson, R Skog, H Spaak, G Gus, D Druta, and M Hafeez. Explicit Trusted Proxy in HTTP/2.0. *IETF Internet-Draft; Intended status: Standards Track*, 2014.

[54] John Ryan McGarvey. Authentication method to enable servers using public key authentication to obtain user-delegated tickets, November 4 2003. US Patent 6,643,774.

[55] D McGrew, Dan Wing, Y Nir, and P Gladstone. TLS proxy server extension. *IETF Internet-Draft Intended status: Informational*, 2012.

[56] David McGrew and John Viega. The Galois/counter mode of operation (GCM). *NIST Modes Operation Symmetric Key Block Ciphers*, 2005.

[57] Gennady Medvinsky, Nir Nice, Tomer Shiran, Alexander Teplitsky, Paul Leach, and John Neystadt. Authentication delegation based on re-verification of cryptographic evidence, June 9 2015. US Patent 9,055,107.

[58] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the S in HTTPS. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 133–140. ACM, 2014.

[59] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 199–212. ACM, 2015.

[60] C. Nikolouzakis. Encrypted traffic grows 40 percent post edward snowden NSA leak. `http://www.sinefa.com/blog/encrypted-traffic-grows-post-edward-snowden-nsa-leak.`

[61] Yoav Nir. A method for sharing record protocol keys with a middlebox in TLS. *IETF Internet-Draft Intended status: Standards Track*, 2012.

[62] M Nottingham. Problems with proxies in HTTP. *IETF Internet-Draft Intended status: Informational*, 2013.

[63] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[64] Roberto Peon. Explicit proxies for HTTP/2.0. *IETF Internet-Draft Intended status: Informational*, 2012.

[65] E Rescorla and A Schiffman. The Secure Hypertext Transfer Protocol. 1999.

[66] Jesse Abraham Rothstein, Arindum Mukerji, David D Schmitt, and John R Hughes. Accessing SSL connection data by a third-party, July 15 2014. US Patent 8,782,393.

[67] R Sandvik. Security vulnerability found in Cyberoam DPI devices (CVE-2012-3372). 2012.

[68] Stuart E Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. The emperor's new security indicators. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 51–65. IEEE, 2007.

[69] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[70] Vyas Sekar, Sylvia Ratnasamy, Michael K Reiter, Norbert Egi, and Guangyu Shi. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 2011.

[71] Karen Seo and Stephen Kent. Security architecture for the Internet protocol. *IETF RFC 7619*, 2005. Category: Standards Track, ISSN 2070-1721.

[72] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[73] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 213–226. ACM, 2015.

[74] George J Silowash, Todd Lewellen, Daniel L Costa, and Lewellen. Detecting and preventing data exfiltration through encrypted web sessions via traffic inspection. 2013.

[75] Douglas Stebila and Nick Sullivan. An analysis of TLS handshake proxying. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 279–286. IEEE, 2015.

[76] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *USENIX security symposium*, pages 399–416, 2009.

[77] Verizon RISK Team. 2015 data breach investigations report. 2015.

[78] Steven Tuecke, Von Welch, Doug Engert, Laura Pearlman, and Mary Thompson. Internet X. 509 public key infrastructure (PKI) proxy certificate profile. Technical report, 2004.

[79] Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Nicholas Weaver, and Vern Paxson. Beyond the radio: Illuminating the higher layers of mobile networks. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 375–387. ACM, 2015.

[80] Nicholas Weaver, Christian Kreibich, Martin Dam, and Vern Paxson. Here be web proxies. In *International Conference on Passive and Active Network Measurement*, pages 183–192. Springer, 2014.

[81] Ofir Weisse, Timothy Trippel, and Jeremy Erickson. Ubicrypt: Making ubiquitous encryption compatible with enterprise security. 2015.

[82] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of caching strategies in modern cellular backhaul networks. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 319–332. ACM, 2013.

[83] Xing Xu, Yurong Jiang, Tobias Flach, Ethan Katz-Bassett, David Choffnes, and Ramesh Govindan. Investigating transparent web proxies in cellular networks.

In *International Conference on Passive and Active Network Measurement*, pages 262–276. Springer, 2015.

[84] K Zetter. The feds cut a deal with in-flight wi-fi providers, and privacy groups are worried. *Wired Magazine*, 2014.

Chapter 7

Appendix

7.1 TLS Message Header Values

Listing 7.1: TLS content type headers

```
CHANGECIPHERSPEC        20   (x‘14’)
ALERT                   21   (x‘15’)
HANDSHAKE               22   (x‘16’)
APPLICATION_DATA        23   (x‘17’)
```

Listing 7.2: TLS version header values

```
SSL3            x‘0300’
TLS1            x‘0301’
TLS1.1          x‘0302’
TLS1.2          x‘0303’
```

Listing 7.3: TLS handshake message format

```
Byte   0   TLS record type = 22
Bytes 1-2  TLS version
Bytes 3-4  TLS Length of data in the record (excluding the
    header itself).
Byte   5   TLS Handshake type (Value 0 for Client/Value 1
   for Server)
Bytes 6-8  TLS Length of data to follow in this record
Bytes 9-n  TLS Command-specific data
```

Listing 7.4: Format of a TLS record

```
Byte   0    TLS record type
Bytes 1-2   TLS version (major/minor)
Bytes 3-4   Length of data in the record (excluding the
   header itself).
The maximum SSL supports is 16384 (16K).
```

7.2 Entities and definitions

The following terms are used in paper and are defined as follows:

1. User: A user is the human being who controls and initiates the use of an user-agent/client.

2. User-agent/Client: The client is a program run in a computer by the user and is responsible for starting a TLS/SSL session with a service provider/server. Client and user-agents are synonyms. A client is also one of the endpoints in a TLS/SSL connection.

3. Server/Service-provider: The server is a networking computer that completes the requests of a client and responds back with data. A server is the other endpoint in a TLS/SSL connection. Service-provider and server are synonyms.

4. Middlebox/Proxy: A middlebox is a generalized term for a networking device that intercepts traffic between a client and a server and uses it for purposes like inspection, manipulation, packet forwarding, caching, compression etc. Middlebox and proxy are synonyms for the purpose of this paper.

5. Transparent proxy: A proxy that uses Split TLS. A networking device that intercepts the traffic between a client and a server. This proxy acts on behalf of the client by adding a trusted root certificate on the user's device. Such a proxy is transparent to both the end points during a TLS/SSL session. The user himself may install the certificate on the device, an adversary may trick the user into installing the certificate or the certificate is already present in the device installed by the manufacturer.

6. Reverse proxy: A proxy interposed by the server (such as gateway or portal) is called as a reverse proxy.

7. End-to-end encryption: Encryption of data between two entities with no intermediates in the connection.

8. User-consent/user-permissions: The explicit grant provided by an user resulting in the trust of a particular component. The user may or may not be aware of this grant. User consent and user-permissions are synonyms.

9. Middlebox service: The functionality done by the middlebox.

7.3 Machine configuration for Triraksha implementation

Configuration for client and middlebox machine: Intel(R) Core(TM) i7-2000 2.3 Ghz, 1 Tb HDD, 8 Gb RAM. Speed for USB to ethernet adapter: USB 2.0 to 100 Mb/sec.

7.4 Definitions for Curl APIs

We leverage Curl's inbuilt timing API's to record logistics for one Curl request. The following APIs are used for our evaluation of Triraksha.

1. time_namelookup: The time, in seconds, it took from the start until the name resolving was completed.

2. time_connect: The time, in seconds, it took from the start until the TCP connect to the remote host (or proxy) was completed.

3. time_appconnect: The time, in seconds, it took from the start until the SSL/TCP/etc connect/handshake to the remote host was completed.

4. time_pretransfer: The time, in seconds, it took from the start until the file transfer was just about to begin. This includes all pre-transfer commands and negotiations that are specific to the particular protocol(s) involved.

5. time_starttransfer: The time, in seconds, it took from the start until the first byte was just about to be transferred. This includes time_pretransfer and also the time the server needed to calculate the result.

6. time_total: The total time, in seconds, that the full operation lasted. The time will be displayed with millisecond resolution.

7. size_header: The total amount of bytes of the downloaded headers.

8. size_download: The total amount of bytes that were downloaded.

7.5 Definitions for Chrome's HAR file

We leverage the Chrome developer tools to record logistics for web requests made with the browser. The definitions for the parameters recorded are:

1. Number of requests: The number of requests made to the principal webserver.

2. Wait time: This is the amount of time waiting for the server to respond.

3. Recieve time: This is the amount of time used for the server to transfer the required information to the client.

4. On load time: Total time taken for the page to be fully loaded, inclusive of AJAX calls or any REST calls from Javascript to populate data on external server.

5. SSL time: The time it took for the SSL handshake to complete.

6. Uploaded data: The total amount of bytes of the uploaded requests.

7. Downloaded data: The total amount of bytes that were downloaded from the response.