RUHR-UNIVERSITÄT BOCHUM

RUB

Bachelor's Thesis in IT Security

# Detecting TLS Interception in the Wild

Okan Saracbasi

SOFTWARE SECURITY

Bachelor's Thesis in IT Security

# Detecting TLS Interception in the Wild

| | |
|---|---|
| Author: | Okan Saracbasi |
| Supervisor: | Prof. Dr. Kevin Borgolte |
| Advisor: | Dr. Talha Paracha |
| Submission Date: | March 4, 2025 |

SOFTWARE SECURITY

## Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich erkläre mich des Weiteren damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

## Statutory Declaration

I hereby declare that I have not submitted this thesis in this or similar form to any other examination at Ruhr University Bochum or any other institution or university to obtain an academic degree.

I confirm that this thesis is solely my own work, and that I have not used any sources other than those stated. Any and all passages taken from other sources, verbatim or otherwise, have been appropriately marked and correctly cited. This also includes graphics, drawings, sketches, and any other components of this work.

I agree that the digital version of this thesis will be used to check it for plagiarism.

The German version of this declaration is legally binding.

| | |
|---|---|
| Ort, Datum / Place, Date | Okan Saracbasi |

# Abstract

Identifying instances of Transport Layer Security (TLS) being intercepted in the Wild is crucial for ensuring secure Internet communications. This thesis concentrates on the implementation of techniques for recognizing possible attempts at interception through the use of both legacy and modern cryptographic technologies.

A client was created to utilize two distinct versions of OpenSSL to establish TLS connections with a server. These connections test various cipher suites, including weak export-grade ciphers. Two different configurations were used on the server side: one utilized a modern version of OpenSSL, while the other was set up to support weak, insecure ciphers. These servers log requests from the client that are unique and identifiable. Through the use of VPN services such as NordVPN, the client was deployed across various geographical locations using 25 different VPN configurations, thereby simulating connections from multiple vantage points. This approach replicates real-world circumstances and enhances the likelihood of detecting interception attempts.

By examining server logs, the experiment seeks to pinpoint patterns that suggest interception, including unique request retransmissions or alterations in the ClientHello, such as the downgrading of cipher suites. Additionally, the experiment leverages iframe paths to detect if intercepted unique requests are subsequently accessed or if the iframe content derived from the unique request is accessed, providing further evidence of potential TLS interception.

In the evaluation phase, analysis of the server logs from over 22,000 connections revealed a consistent match between the TLS configurations embedded in the unique requests and those observed at the server. Notably, no retransmissions of unique requests or unauthorized accesses to the corresponding iframe paths were detected.

# Contents

# 1 Introduction

One crucial technology for protecting online communication is Transport Layer Security (TLS). Data sent between clients and servers is encrypted to prevent eavesdropping by a third party [8]. TLS has become the de facto standard for securing a vast range of applications, including web browsing and email. In fact, TLS underpins HTTPS, which is used by approximately 90% of all websites, ensuring that sensitive information is transmitted securely across the globe [24].

However, the security of TLS can be compromised by interception, where a third party intercepts and decrypts the communication. Interception might occur for a variety of reasons. Malicious actors may exploit vulnerabilities to eavesdrop or manipulate data, but interception can also arise in more benign contexts. For example, governments or network administrators might intentionally intercept traffic for surveillance or security monitoring.

While modern TLS versions are generally considered secure, the possibility of unknown vulnerabilities or attacks exploiting older versions remains [2]. This thesis assumes that an attacker may possess knowledge of such a vulnerability, allowing us to explore the potential impact of TLS interception and investigate methods for its detection.

This thesis aims to explore and implement multiple techniques for detecting TLS interception in real-world scenarios, utilizing different methods. One technique involves using client-side code to make numerous TLS requests that differ in their settings and using server-side code to log these requests in a way that would help detect interceptions. Additionally, the client is deployed in various parts of the world using VPN.

We will investigate various techniques for detecting interception.

This research aims to contribute to a better understanding of TLS interception techniques and provide methods to detect TLS interceptions.

## 1.1 Motivation

In today's digital landscape, Transport Layer Security has become an indispensable protocol for securing communications across the internet. As highlighted earlier, history has repeatedly demonstrated that attackers can potentially eavesdrop on and decrypt TLS communications. This inherent vulnerability motivates this Bachelor thesis, which aims to explore and evaluate methods for detecting TLS interception.

We will introduce detection techniques. A key focus will be on the practical implementation of these methods. By implementing promising techniques, we can gain insights into their real-world applicability and potential impact on existing applications. This implementation-driven approach will help to create practical methods that will allow interception detection in the real-world.

### Language Model Assistance

In the preparation of this thesis, the author has utilized large language models (LLMs), including ChatGPT, to assist with translation, sentence structuring, and overall improvements in readability. These tools were employed solely to enhance the clarity and coherence of the language used throughout the document. All technical content, analysis, and original contributions remain the independent work of the author.

# 2 | Related Work

Cloudflare's blog post ”Monsters in the Middleboxes” discusses the prevalence of HTTPS interception on the internet and introduces two tools developed by Cloudflare to detect such practices: MITMEngine, an open-source library that uses TLS fingerprinting to detect interceptions, and MALCOLM, a dashboard displaying metrics about HTTPS interception observed on Cloudflare's network [14].

Xing et al. [26] conducted a global measurement study to investigate the phenomenon of ”traffic shadowing,” where on-path observers capture network traffic and later generate unsolicited requests based on the observed data. The study employed a VPN-based measurement platform with over 4,000 vantage points to send decoy traffic across various protocols (DNS, HTTP, TLS) and capture resulting unsolicited requests. The researchers found that DNS queries were most susceptible to shadowing, with a significant portion originating from IP addresses flagged as malicious. Notably, the study observed unsolicited requests even after a 10-day interval, indicating potential long-term retention of user data.

The findings of Xing et al. [26] highlight the privacy implications of traffic shadowing and emphasize the need for further research to understand the motivations and mechanisms behind this behavior. While my thesis focuses on TLS interception and exploitation of vulnerabilities, Xing et al. [26] reveals that passive forms of interception like traffic shadowing can also compromise user privacy. Both studies utilize VPNs for measurement, albeit with different setups and objectives. Finally, Xing et al. [26] finding that a significant portion of traffic observers were located in China raises questions about the role of such shadowing in Internet censorship and surveillance, a topic relevant to the broader discussion of TLS interception and its implications.

Ćurguz, in [10], analyzes several vulnerabilities of the SSL/TLS protocol, focusing on the handshake process. The author examines attacks such as the cipher suite rollback attack, version rollback attack, and key exchange algorithm confusion. These attacks exploit weaknesses in the negotiation and authentication mechanisms, allowing adversaries to force a connection to use weaker or even null encryption schemes.

Ćurguz highlights that when the handshake process is compromised, the resulting security parameters may not adequately protect data confidentiality and integrity. The author emphasizes that the ability of an attacker to manipulate the handshake process could, in practice, be exploited to intercept secure communications. This underscores the critical importance of robust defenses in SSL/TLS implementations to ensure that the negotiated parameters truly safeguard the transmitted data.

An article from Security.org [22] explores use cases for internet censorship, defining it as the control over what can be accessed, published, or viewed online. The motivations for censorship include:

- **Political censorship:** Suppressing information or opinions that are critical of a government or political party.

- **Workplace censorship:** Restricting access to inappropriate content to increase productivity.
- **Safety censorship:** Blocking access to inappropriate content to protect children.

The article also describes various methods used to implement censorship:

- **DNS tampering:** Manipulating DNS records to redirect users to incorrect or blocked websites.
- **IP address blocking:** Denying access to specific websites or online services based on their IP addresses.
- **Keyword filtering:** Blocking access to content that contains specific keywords or phrases.
- **Packet filtering:** Selectively blocking or discarding network packets based on their content or other characteristics.
- **Traffic shaping:** Prioritizing or limiting the bandwidth allocated to specific types of traffic, which can slow down or block access to certain websites or services.
- **Port number blocking:** Blocking access to specific ports, thereby preventing communication with certain applications or services.

Among these methods keyword filtering, packet filtering and traffic shaping are particularly relevant to our work because they typically require TLS interception to be effectively implemented.

# 3 | Background

## 3.1 Transport Layer Security

Transport Layer Security (TLS) is a protocol designed to ensure secure communication between two entities by providing confidentiality, integrity, and authentication. Confidentiality ensures that the transmitted data remains readable only to the intended parties. Integrity mechanisms ensure that changes to the data during transmission are detected. Authentication verifies the identity of the communicating parties, primarily the server, and optionally the client.

The TLS protocol is structured around two primary components: the handshake protocol and the record protocol. The handshake protocol establishes the secure connection by authenticating the participating entities, negotiating cryptographic parameters and encryption methods. This protocol is designed to be resistant to manipulation, preventing attackers from influencing the selection of cryptographic parameters. The record protocol then uses the established parameters to protect the network traffic, ensuring confidentiality through encryption and maintaining integrity through data verification methods [21].

### 3.1.1 TLS Versions and Evolution

The evolution of Transport Layer Security (TLS) began with the development of Secure Sockets Layer (SSL) by Netscape in 1995. SSL 1.0 and 2.0 were not publicly released due to security flaws. However, SSL 3.0, introduced in 1996, addressed many of these vulnerabilities and became widely adopted. In January 1999, the Internet Engineering Task Force (IETF) published RFC 2246, introducing TLS 1.0 as a standardized protocol to ensure interoperability across web browsers and servers. Subsequent versions, TLS 1.1 and TLS 1.2, were released in 2006 and 2008, respectively, offering support for more advanced cipher suites and improved security features. Despite these advancements, earlier versions of TLS, including 1.0 and 1.1, lacked support for modern cryptographic algorithms and were susceptible to various attacks. In 2018, TLS 1.3 was introduced, representing a significant overhaul of the protocol. This version removed obsolete features and enhanced security The National Institute of Standards and Technology (NIST) recommended organizations adopt TLS 1.3 to ensure robust encryption for online interactions [13].

### 3.1.2 TLS Handshake

The TLS Handshake aims to authenticate both the server and, if desired, the client. Additionally, during the handshake, cryptographic settings and encryption techniques are negotiated. From TLS 1.0 to TLS 1.2, the TLS handshake follows a similar structure. However, TLS 1.3 introduces several key changes to improve security and performance. These include the removal of outdated cryptographic algorithms such as RSA key exchange and static Diffie-Hellman, the elimination of handshake renegotiation to prevent downgrade attacks, and a shift to forward secrecy by mandating ephemeral key exchanges. Additionally, TLS 1.3 removes unnecessary round trips.

#### 3.1.2.1 TLS 1.2 and Earlier Handshake

The TLS Handshake is a series of messages exchanged between a client and a server to establish a secure connection. It ensures authentication, key exchange, and agreement on cryptographic parameters. The handshake process in TLS 1.2 and earlier follows a structured sequence of message exchanges, as specified in [11].

During the handshake, the client and server may use the Diffie-Hellman key exchange protocol to securely negotiate a shared secret key for encryption.

The process begins with the **ClientHello** message, where the client proposes supported TLS versions, cipher suites, and a randomly generated value. The server responds with a **ServerHello** message, selecting a compatible TLS version and cipher suite.

To authenticate itself, the server provides a **Certificate** message, and in some cases, a **ServerKeyExchange** message is sent to facilitate key agreement. If mutual authentication is required, the server issues a **CertificateRequest** before concluding its part of the handshake with **ServerHelloDone**.

The client then verifies the server's certificate and proceeds with the **ClientKeyExchange** message, transmitting key exchange parameters. If client authentication is requested, it also sends its **Certificate** and a **CertificateVerify** message.

Finally, both parties confirm the completion of the handshake by exchanging **ChangeCipherSpec** and **Finished** messages. This establishes an encrypted channel, ensuring confidentiality and integrity for subsequent communication.

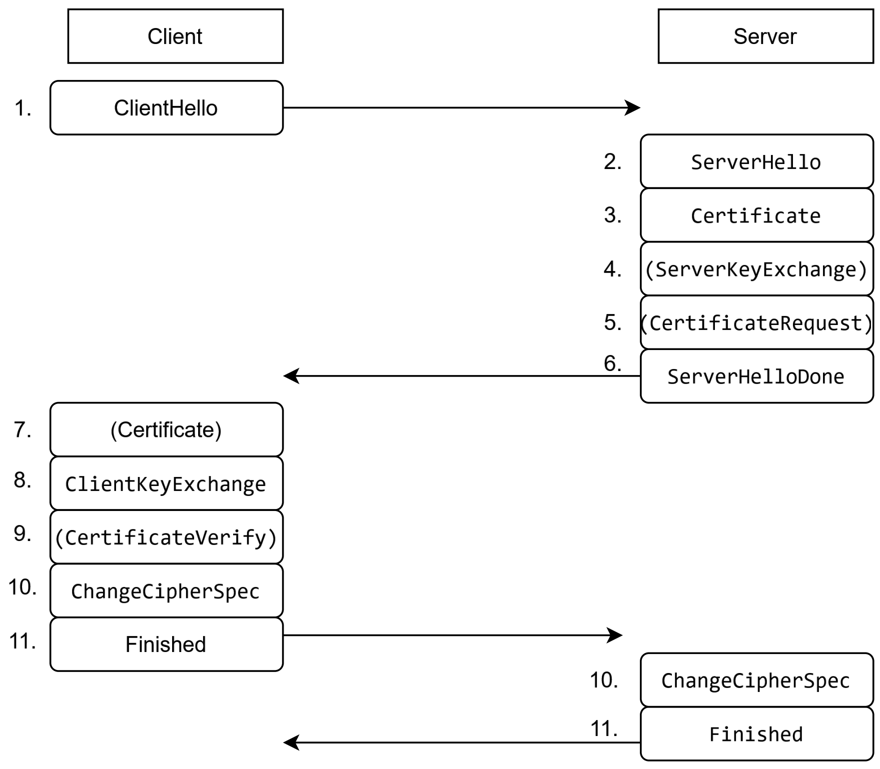

**Figure 3.1:** Overview of the TLS 1.2 Handshake.

1. **Client Hello:** The client initiates the handshake by sending a `ClientHello` message. This message contains:
   - The highest TLS protocol version supported by the client.

- A random client-generated value, which will be used in the key derivation process.
- Optionally, a session ID, which can be used to resume a previous session if the client wants to reuse already negotiated parameters.
- A list of cipher suites supported by the client, ordered by preference.
- A list of supported compression methods.
- An extensions list, containing optional extensions that the client wishes to use.

2. **Server Hello:** The server responds with a `ServerHello` message. This message contains:
   - The selected TLS protocol version, chosen from the client's list of supported versions.
   - A server-generated random value.
   - Optionally, a session ID.
   - The selected cipher suite, chosen from the client's list.
   - The selected compression method.
   - The extensions list, indicating which extensions the server supports.
3. **Certificate:** The server sends its certificate chain. This allows the client to authenticate the server's identity. The certificate must be appropriate for the chosen cipher suite and key exchange algorithm. It should be signed by a trusted certificate authority and contain the server's public key.
4. **Server Key Exchange:** This message is optional and sent only if the server needs to provide additional data for the key exchange process. This might be necessary if the chosen cipher suite requires parameters that are not included in the server's certificate.
5. **Certificate Request:** This message is optional. The server can request a certificate from the client if client authentication is required.
6. **Server Hello Done:** The server signals the end of its messages with this message.
7. **Client Certificate:** If the server requested a certificate, the client sends its certificate chain in this message.
8. **Client Key Exchange:** The client sends the key exchange data, which allows both parties to compute the premaster secret. The exact content of this message depends on the chosen key exchange algorithm.
9. **Certificate Verify:** This message is sent if the client provided a certificate. It contains a digital signature, which allows the server to verify that the client possesses the private key corresponding to the certificate.
10. **Change Cipher Spec:** Both client and server send this message to indicate that they will switch to the negotiated cipher suite and encryption keys for the rest of the communication.
11. **Finished:** Both client and server send this message to verify that the key exchange and authentication were successful.

#### 3.1.2.2 TLS 1.3 Handshake

The TLS 1.3 handshake introduces significant changes compared to TLS 1.0-1.2 [9, 21]. These changes aim to enhance security, improve performance, and reduce handshake latency.

The **ClientHello** message now includes parameters for calculating the premaster secret. This is possible because the client can anticipate the server's preferred key exchange method based on the reduced set of supported cipher suites. This optimization minimizes the need for multiple rounds of key exchange parameter negotiation.

After receiving the **ClientHello**, the server can immediately compute the master secret. The server then sends a **ServerHello** message, which includes the following:

- The selected cipher suite.
- The server's certificate.
- A digital signature for authentication.
- The server random value.

Additionally, the server sends the **Finished** message in the same step, as it has already computed the master secret. The client can then use the information from the **ServerHello** to compute the master secret and send its own **Finished** message.

Another major feature of TLS 1.3 is support for **0-Round Trip Time (0-RTT) resumption**. This allows a client to send application data in the very first message of a resumed session.

TLS 1.3 also removes several outdated or insecure features and algorithms. The following elements are no longer supported [5]:

- RC4, 3DES and Camellia ciphers
- CBC mode ciphers
- SHA-1 and md5 hash function
- RSA key exchange
- Renegotiation
- Compression

### 3.1.3 The TLS Record Layer

The TLS Record Layer is responsible for ensuring the confidentiality, integrity, and authentication of application data during transmission. It operates beneath the handshake protocol and applies the cryptographic parameters negotiated during the handshake to secure the communication [11].

The responsibilities of the Record Layer include [11, 21]:

- **Fragmentation:** Incoming data from the application layer is segmented into TLS records with a maximum payload size of 16,384 bytes.
- **Compression:** Prior to encryption, the data can be compressed to reduce its size. However, starting from TLS 1.3, compression has been removed to prevent attacks like CRIME and BREACH that exploit compression to recover secret information.
- **Encryption:** The fragmented data is encrypted using the negotiated cipher suite and keys. Encryption ensures that the transmitted data remains confidential.
- **Message Authentication:** A Message Authentication Code (MAC) is computed and appended to the record to ensure data integrity. The MAC allows the receiver to verify that the data has not been changed during transmission. In TLS 1.3, authenticated encryption (AEAD) is used to integrate encryption and authentication into a single operation.

- **Transmission:** The encrypted and authenticated record is sent over the network to the recipient, who then can decrypt and verify it.

## 3.2 TLS Interceptions

TLS interception generally refers to the situation where a third party is capable of decrypting a TLS-protected communication between a client and a server or manipulating the transmitted data. In this section, different methods of how a TLS connection could potentially be intercepted are introduced.

### 3.2.1 Man-in-the-Middle Attack

A Man-in-the-Middle (MitM) attack occurs when a third party intercepts the communication between a client and a server, allowing them to read, modify, or inject messages. To position themselves between the client and server, attackers can employ various techniques, such as [6]:

- **ARP poisoning:** This technique exploits the Address Resolution Protocol (ARP) to redirect traffic intended for the server to the attacker's machine. This is typically effective within a local network [20].
- **IP spoofing:** The attacker masquerades as the server by forging its IP address in outgoing packets. To receive the responses, the attacker needs to be on the same network or on the path between the client and the server.
- **DNS spoofing:** The attacker manipulates Domain Name System (DNS) records to redirect the client to a malicious server under their control.
- **On-path attacks:** An attacker who controls a network device on the communication path between the client and server, such as a router or firewall, can intercept and manipulate traffic. This privileged position allows them to observe, modify, or block communication.

For a successful MitM attack, the attacker typically needs a forged certificate that the client will accept or must obtain a legitimate certificate for the server through fraudulent means.

In a typical MitM attack scenario [6]:

1. The attacker intercepts the client's initial connection request to the server.

2. The attacker then impersonates the server, presenting their own certificate to the client and establishing a TLS connection.

3. Simultaneously, the attacker establishes a separate TLS connection to the legitimate server, acting as a client.

4. The attacker can then relay messages between the client and server, potentially reading, modifying, or injecting data into the communication stream.

This effectively allows the attacker to eavesdrop on the communication and manipulate data. Additionally, attackers can exploit vulnerabilities in the implementation or configuration of TLS to compromise the security of the connection.

### 3.2.2 Weak Cipher Suites

TLS supports a variety of cipher suites, but not all of them are secure. For example, export cipher suites were intentionally weakened during the crypto wars of the 1990s, when U.S. government regulations imposed restrictions on the export of strong cryptography. Their weaknesses stem from the use of older cryptographic algorithms like MD5 and significantly limited key lengths [23].

Older ciphers like RC4, DES, and 3DES are also regarded as weak in addition to export ciphers. An attacker may be able to decrypt network traffic if a client and server utilize a weak cipher suite and they are able to intercept a sufficiently large volume of ciphertext. Furthermore, as the FREAK attack illustrates, an attacker may attempt to force the usage of weak ciphers [19].

The FREAK attack exploits a vulnerability in OpenSSL to downgrade the TLS connection to use export-grade cipher suites. The attack works as follows [19]:

1. The client sends a `ClientHello` message with RSA cipher suites.

2. The attacker, acting as a MitM, modifies the request to include export-grade RSA cipher suites.

3. The server responds with a 512-bit export-grade RSA key.

4. Due to a vulnerability in certain OpenSSL builds and different browsers the client accepts the weak key.

5. The attacker can then recover the decryption key by factoring the RSA modulus, which is feasible with a 512-bit key.

This allows the attacker to decrypt the communication and potentially steal sensitive information.

### 3.2.3 The BEAST Attack

The BEAST (Browser Exploit Against SSL/TLS) attack, discovered in 2011 by Thai Duong and Juliano Rizzo, targeted a vulnerability in the way TLS 1.0 handled cipher block chaining (CBC) mode encryption. This attack allowed an attacker to decrypt encrypted cookies and potentially hijack user sessions [27].

The BEAST attack exploits the fact that in CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This creates a dependency between blocks, which an attacker can exploit. The attack works as follows:

1. The attacker positions themselves as a man-in-the-middle (MitM) between the client and the server.

2. The attacker injects malicious JavaScript code into the client's browser, for example, through a compromised website.

3. The JavaScript code monitors the encrypted traffic between the client and the server. It also manipulates the client's requests, injecting arbitrary data into the plaintext.

4. By carefully crafting the injected data and observing the resulting ciphertext, the attacker can deduce the value of the previous ciphertext block.

5. By repeating this process block by block, the attacker can decrypt the entire message, including sensitive information like cookies. [27]

The goal of the BEAST attack is to decrypt encrypted cookies and potentially hijack active user sessions.

### 3.2.4 Bleichenbacher's Attack and ROBOT

Bleichenbacher's attack, which was later revisited and expanded upon in the Return Of Bleichenbacher's Oracle Threat (ROBOT) attack, is a type of adaptive chosen-ciphertext attack against RSA

PKCS #1 v1.5 encryption. The original attack, introduced by Daniel Bleichenbacher in 1998, exploited the ability to distinguish between correctly and incorrectly formatted PKCS #1 v1.5 padding after decryption. This allowed an attacker to use a vulnerable server as an oracle to decrypt RSA ciphertexts or forge signatures without knowing the private key [4].

#### 3.2.4.1 Bleichenbacher's Attack (1998)

This original attack relied on the server's behavior when processing invalid ciphertexts. If the server responded differently to valid and invalid ciphertexts, an attacker could exploit this information to iteratively refine their knowledge about the plaintext. By sending a variety of ciphertexts and observing the server's responses, the attacker could eventually recover the entire plaintext [4].

#### 3.2.4.2 ROBOT Attack (2018)

The ROBOT attack, presented by Hanno Böck, Juraj Somorovsky, and Craig Young in 2018, demonstrated that Bleichenbacher's attack was still applicable to modern TLS implementations, despite countermeasures in the TLS standard [4]. ROBOT identified new side-channels that allowed attackers to distinguish between valid and invalid padding values, such as:

- TCP resets
- TCP timeouts
- Duplicate alert messages

To mitigate these attacks, it is crucial to ensure that servers do not leak any information about the validity of PKCS #1 v1.5 padding. One way to do that is to ensure that servers always respond with the same error message, regardless of the padding validity.

### 3.2.5 The Logjam Attack

The Logjam attack is a man-in-the-middle attack that exploits the ability to downgrade a TLS connection to export-grade cryptography [1]. This attack targets the Diffie-Hellman (DH) key exchange used in TLS and is particularly effective against servers that support outdated "export-grade" DH cipher suites with 512-bit primes.

#### 3.2.5.1 Attack Description

The attack proceeds as follows:

1. **Downgrade:** The attacker intercepts the TLS handshake and modifies the client's `ClientHello` message to request only export-grade DH cipher suites.
2. **Handshake Manipulation:** The attacker also modifies the server's `ServerHello` message to indicate that a non-export cipher suite has been selected, deceiving the client.
3. **Key Exchange:** The server sends its Diffie-Hellman parameters with a 512-bit prime. The client, believing it is using a stronger cipher suite, accepts these parameters.
4. **Discrete Log Computation:** The attacker, having performed precomputation for common 512-bit DH groups, can quickly compute the discrete log of the server's public key. This allows them to derive the shared secret and the session keys.
5. **Decryption and Impersonation:** The attacker can now decrypt the communication between the client and the server and impersonate either party.

#### 3.2.5.2 Impact and Mitigations

To mitigate the Logjam attack, it is essential to:

- Disable export-grade cipher suites on TLS servers.
- Use sufficiently large DH groups (at least 2048 bits) to prevent feasible discrete log computations.
- Update clients and servers to the latest versions of TLS, which include countermeasures against downgrade attacks.

### 3.2.6 TLS/SSL Stripping

While not a TLS interception method in the strict sense, TLS/SSL stripping is a technique that allows an attacker to circumvent the encryption provided by HTTPS [3]. It was popularized by Moxie Marlinspike in 2009. The attacker exploits the fact that many users do not explicitly type "https://" in the address bar and instead rely on websites to redirect them to the secure HTTPS version.

The attack works as follows:

1. The client attempts to connect to a website using HTTPS.
2. The attacker intercepts the connection and redirects the client to the unencrypted HTTP version of the site.
3. The client, unaware of the downgrade, sends sensitive information such as login credentials in cleartext to the attacker.
4. The attacker establishes a separate HTTPS connection to the server, forwarding the client's data and relaying the server's responses back to the client.

This effectively allows the attacker to eavesdrop on the communication.

Website operators can implement HTTPS Strict Transport Security (HSTS), which forces browsers to always use HTTPS for their site.

## 3.3 Interception Detection Techniques

There are many different methods to detect and prevent TLS interception, each with its own advantages and disadvantages. This section will introduce some of the techniques.

### 3.3.1 Key Pinning

Key pinning is a technique that allows a client to compare the certificate it receives from the server to a previously stored, known certificate. This enables the client to detect a man-in-the-middle (MitM) attack if the attacker uses a forged certificate. There are different variations of key pinning [7]:

1. **With Client History:** The client stores a certificate from a previous connection and compares it to the certificate presented by the server in subsequent connections. If the certificate has changed, it could indicate a MitM attack. This method requires that the initial connection was secure and trustworthy. False positives can occur if the server legitimately changes its certificate. This issue is exacerbated by the increasing use of short-lived certificates, such as those from Let's Encrypt, which necessitate more frequent renewals and increase the likelihood of false positives [12].

2. **With Server:** The server sends certificate attributes to the client, which are unlikely to change, so the client can pin these attributes. For example, the server could provide a set of public keys that must be present in all certificates. This method also assumes that the first connection is trustworthy. False positives are less likely with this approach.

3. **With Preload:** Browser vendors can include pins for specific websites within the browser itself. For example, Google pins certificates for its own domains and others upon request. This method, unlike the previous ones, does not require trusting the first connection, as long as the browser is trusted and installed securely. False positives are unlikely as long as the browser keeps the pins updated.

4. **With DNS:** The DNS-based Authentication of Named Entities (DANE) protocol allows servers to pin their public key in their DNS records, secured by DNSSEC. Clients can use DANE to validate server certificates by retrieving the public key from the DNSSEC-protected record. This method does not require trust on the first connection as long as the DNSSEC record and the nameserver are trusted. False positives are unlikely to occur.

### 3.3.2 Multipath Probing

The following discussion of Multipath Probing is derived from the findings of Clark and van Oorschot [7]. The method of Multipath Probing involves a client obtaining certificates through independent observers distributed across the internet. The client compares these certificates with the one it received directly to ensure there is no local Man-in-the-Middle (MitM) attack. By relying on independent sources, this method significantly reduces the likelihood of false positives. However, this approach is ineffective if an attacker controls a network segment through which the client accesses the independent observers, making it impossible for the client to verify the authenticity of the received certificates through independent means.

## 3.4 Virtual Private Network

A Virtual Private Network (VPN) is a service that creates a secure, encrypted connection over a public network like the Internet. It allows users to establish a protected link between their device and a VPN server, essentially extending a private network across a public one, enabling secure data transmission. VPNs are commonly used to protect privacy, enhance security, and bypass geographical restrictions [18].

VPNs function by using tunneling protocols (e.g., IPsec, SSL/TLS, OpenVPN) to encapsulate and encrypt data at the sending end and decrypt it at the receiving end. This process hides your IP address, encrypts your internet traffic, and routes it through the VPN server, effectively masking your online identity and location . By connecting to a VPN server in a different location, your internet traffic appears to originate from that server. VPNs link the user's client to the IP address of the VPN server [18].

# 4 Methodology

The goal of our experiment is to detect and analyze TLS interceptions in the real world. We achieve this goal by combining a honeypot setup with systematic cipher testing. This experiment aims to detect TLS interceptions and uncover potential vulnerabilities and attack patterns.

This chapter describes the design and strategy of our experiment. It outlines the components of the setup, the data collection process, and the detection mechanisms used.

## 4.1 Overview of Approach

This experiment is structured to simulate real-world TLS communication and to check for TLS interception. The basic idea is to use a client-server architecture where:

- A client initiates TLS connections to a server using various TLS configurations and cipher suites, including weak ones. In doing so, the client sends unique requests to the server.
- The server logs requests that are structured like the unique requests, along with information about the request that might help to identify an interception.
- VPN connections are used to simulate geographically different connections and to examine interception attempts from diverse network paths.

## 4.2 Experimental Setup

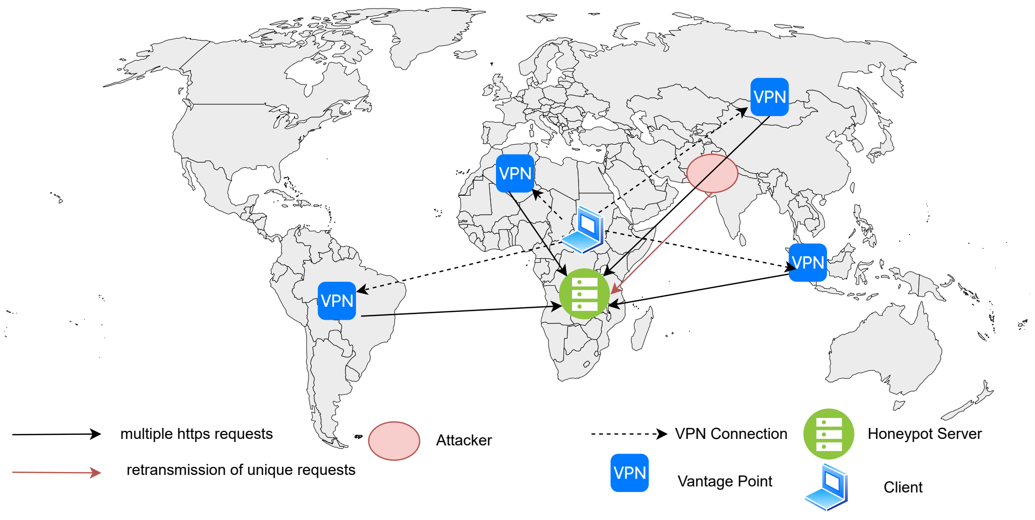

**Figure 4.1:** Experimental Setup. Map source: Simplemaps.com (2020), MIT License.

### 4.2.1 Client

The client is designed to connect with a wide range of TLS configurations. To achieve this, two different OpenSSL versions are used. One OpenSSL instance is an older version with support for export-grade ciphers and other weaker ciphers that are not available in more modern versions of OpenSSL. The second instance of OpenSSL is a more modern version to also support modern ciphers.

The client attempts to establish TLS connections with the server using all supported cipher suites that each OpenSSL instance offers. The client also sends a unique HTTPS request to the server, structured as follows:

**www.destination.de/path/<AES-encrypted: time&tlsversion&cipher>**

This unique path contains the expected TLS configuration for the connection and is very unlikely to be accessed randomly. This characteristic will help in identifying any retransmissions or manipulations of the connection.

### 4.2.2 Server

The server setup uses two different OpenSSL instances. One instance uses an older version of OpenSSL configured to support export-grade and other weak cipher suites. The other instance uses a more modern version of OpenSSL to handle modern ciphers that are not supported by the older version. Both instances work under the same IP and are be reachable through the HTTPS port 443. To achieve this, a single Nginx server is configured as a TCP passthrough that listens on port 443 and forwards requests based on Server Name Indication (SNI).

Utilizing both modern and legacy OpenSSL instances enables a comprehensive evaluation of potential TLS interception vulnerabilities. This approach allows us to test a broad spectrum of cipher suites, including export-grade ciphers. Furthermore, employing a TCP passthrough guarantees the preservation of the original TLS handshake.

For each unique HTTPS request, the server replies with an iframe that includes a unique path associated with the original request. This path aids in identifying any intercepted connections.

The server is configured to log the HTTPS requests and information about the connection, such as the TLS version, cipher suite, and client IP address.

### 4.2.3 VPNs

To simulate connections from different locations, we use NordVPN with OpenVPN configurations. This allows the client to connect to the server through different network paths, increasing the potential for encountering an attacker. For the experiment, we utilize 25 OpenVPN configurations from NordVPN, listed in Chapter 5. It is important to note that these 25 configurations yield more than 25 distinct IP addresses. The use of multiple VPN configurations is critical for replicating the diversity of real-world network conditions, as interception techniques may vary across different geographic regions and network infrastructures. This variation is often due to legal and regulatory reasons, as different countries and jurisdictions have different laws and regulations regarding internet surveillance and data interception. Some countries may permit or even mandate interception for law enforcement or national security purposes, while others may have strict privacy laws that prohibit or limit interception. By testing connections from various vantage points, the experiment aims to capture a more comprehensive picture of potential TLS interception vulnerabilities.

## 4.3 Collection of Data

In this experiment, we collect the following data from the connections:

- The accessed path from the HTTP request, which contains information about the date/time, used TLS version, and cipher suite.
- The client IP address.
- The virtual host (SNI), which indicates which server the client connected to.
- The User-Agent.
- The effectively used TLS version.
- The effectively used cipher suite.
- A timestamp indicating when the message arrived.
- The unique iframe path for each HTTPS request.
- Accessed iframe paths, with the corresponding path that sent the iframe path.

## 4.4 Detection Strategy

Several indicators can reveal potential TLS interceptions.

**Re-accessed Unique Path:** If a unique path is accessed a second time, TLS interception is likely. The probability of random bots accessing these long, encryption-derived paths is extremely low, considering their length, the information they contain, and the use of random nonces.

**TLS Version and Cipher Suite Discrepancies:** Logged information about TLS versions and cipher suites provides another strong indicator. The unique path contains details about the TLS version and cipher suite the client intends to use. If these differ from the recorded TLS version or cipher suite actually used for the connection, it suggests an attacker is attempting to interfere with the TLS handshake. This is because the client should only use the cipher suite from the path for the handshake, which should force the server to choose the cipher that is also encoded in the path. Two server-side scripts (see Appendix A.6) automate this analysis: one checks for TLS version discrepancies, and the other compares the cipher suite.

**Iframe Path Access:** The iframe path offers another detection method. The client is configured to request only the main path; the iframe and its path are part of the server's response. Direct access to the iframe path indicates a leak or interception, as its length and construction make random access highly improbable. Any access to the iframe path, whether directly or via a browser accessing the main path, will be logged by the server. This is because the client itself never requests the iframe.

**Timestamp Analysis:** Timestamps can also reveal interceptions. An unusually long request time compared to other requests from the same vantage point suggests potential TLS interception. This assumes a reasonably consistent network environment for comparison.

## 4.5 Rationale for Expected Access

- **Curiosity:** Attackers, driven by their inquisitive nature, often explore all available avenues within a target system. This inherent curiosity extends to seemingly insignificant elements like the iframe path, which may be accessed simply to gain a more complete understanding of the website's structure or potential vulnerabilities.

- **Automated Scanning:** The widespread use of automated tools by attackers for vulnerability scanning and information gathering further increases the likelihood of the iframe path being accessed. These tools systematically follow all links and paths, including the iframe path, regardless of their perceived importance. This indiscriminate approach increases the likelihood of the iframe path being accessed, even without specific targeting by the attacker.

## 4.6 Rationale for the Detection Method

The rationale behind this method is that it can help detect TLS interceptions through multiple indicators. With this honeypot setup, it is possible to identify an attacker regardless of the specific attack technique employed. This method relies on the assumption that an attacker, if present, will be curious enough to access either the unique request path or the corresponding iframe after successfully intercepting the TLS connection.

Furthermore, the technique allows for the detection of manipulations in the ClientHello message. Since the TLS version and cipher suite information are embedded in the unique path, any discrepancy between the intended values and those observed in the connection may indicate that an attacker has altered the handshake. This is particularly useful for detecting downgrade attacks.

Additionally, the time difference between the timestamp in the unique path and the time at which the request reaches the server may serve as an indicator of TLS interception, if there are significant discrepancies compared to other tested connections.

The use of two different OpenSSL versions—one modern (1.1.1f) and one legacy (1.0.2u)—enables the client to test a wide range of cipher suites, including those known to be insecure. This approach allows for the detection of various types of attackers. An attacker might exploit known vulnerabilities in weak cipher suites to intercept the connection. Conversely, an attacker may also employ unknown methods that are effective against modern cipher suites. Moreover, the inclusion of the `eNULL` cipher suite further assists in detecting interception, as any leakage of a unique path would be evident.

Finally, it is important to note that even if no anomalies are detected, this does not conclusively prove the absence of an attacker—it only indicates that, under the conditions of the experiment, no overt manipulations (such as retransmission of unique paths or unauthorized iframe accesses) were observed.

# 5 Implementation

This chapter provides detailed information about the technical implementation of the experiments described in Chapter 4. It offers insights into the tools, configurations, and scripts used to create the client-server setup. The goal is to describe how the honeypot infrastructure was built to detect TLS interceptions.

## 5.1 Server Setup

The server runs on a virtual machine (VM) with Ubuntu 20.04.6 LTS installed. This setup uses two instances of Nginx version 1.27.3, each handling a different set of cipher suites.

### 5.1.1 First Nginx Instance

The first instance of Nginx is responsible for handling modern cipher suites and acts as a proxy, allowing both Nginx instances to operate under the same IP address.

#### 5.1.1.1 Dependencies and Preparation

To install and configure Nginx, we must first install the required build tools and dependencies:

```
sudo apt update
sudo apt install build-essential zlib1g-dev libpcre3-dev libssl-dev \
    libxslt1-dev libgd-dev
```

#### 5.1.1.2 Configuration

The first instance uses OpenSSL 1.1.1f, which is either pre-installed or can be installed via:

```
sudo apt install openssl
```

We prepared the configuration in the downloaded Nginx source folder using the following command:

```
./configure \
    --prefix=/usr/local/nginx \
    --with-cc-opt='-g -O2 -fno-omit-frame-pointer \
        -mno-omit-leaf-frame-pointer \
        -ffile-prefix-map=/build/nginx-DlMnQR/nginx-1.27.3=. \
        -flto=auto -ffat-lto-objects -fstack-protector-strong \
        -fstack-clash-protection -Wformat -Werror=format-security \
        -fcf-protection \
        -fdebug-prefix-map=/build/nginx-DlMnQR/nginx-1.27.3=/usr/src/nginx-1.27.3 \
        -fPIC -Wdate-time' \
    --with-ld-opt='-Wl,-Bsymbolic-functions -flto=auto -ffat-lto-objects \
        -Wl,-z,relro -Wl,-z,now -fPIC' \
    --conf-path=/etc/nginx/nginx.conf \
    --http-log-path=/var/log/nginx/access.log \
    --error-log-path=stderr \
    --lock-path=/var/lock/nginx.lock \
    --pid-path=/run/nginx.pid \
    --modules-path=/usr/local/nginx/modules \
    --http-client-body-temp-path=/var/lib/nginx/body \
    --http-fastcgi-temp-path=/var/lib/nginx/fastcgi \
    --http-proxy-temp-path=/var/lib/nginx/proxy \
    --http-scgi-temp-path=/var/lib/nginx/scgi \
    --http-uwsgi-temp-path=/var/lib/nginx/uwsgi \
    --with-compat \
    --with-debug \
    --with-pcre-jit \
    --with-http_ssl_module \
    --with-http_stub_status_module \
    --with-http_realip_module \
    --with-http_auth_request_module \
    --with-http_v2_module \
    --with-http_dav_module \
    --with-http_slice_module \
    --with-threads \
    --with-http_addition_module \
    --with-http_flv_module \
    --with-http_gunzip_module \
    --with-http_gzip_static_module \
    --with-http_mp4_module \
    --with-http_random_index_module \
    --with-http_secure_link_module \
    --with-http_sub_module \
    --with-mail_ssl_module \
    --with-stream_ssl_module \
    --with-stream_ssl_preread_module \
    --with-stream_realip_module \
    --with-http_geoip_module=dynamic \
    --with-http_image_filter_module=dynamic \
    --with-http_perl_module=dynamic \
    --with-http_xslt_module=dynamic \
    --with-mail=dynamic \
    --with-stream=dynamic \
    --with-stream_geoip_module=dynamic
```

This configuration enables several important modules:

- `--with-http_ssl_module`: Enables TLS functionality.
- `--with-http_realip_module`, `--with-stream_realip_module`: Ensures the client's real IP address is preserved when the request is proxied internally.
- `--with-stream_ssl_module`, `--with-stream_ssl_preread_module`: Enables stream-based SSL handling, allowing the server to act as a proxy and select the appropriate backend server based on the SNI (Server Name Indication) in the handshake.

We installed the instance using:

```
make
sudo make install
```

We set the installation path to `/usr/local/nginx`.

### 5.1.2 Second Nginx Instance

We configured the second Nginx instance to handle export-grade and other weak cipher suites. It uses OpenSSL 1.0.2u, which supports older and weaker ciphers.

#### 5.1.2.1 Configuration

The configuration process is similar to the first instance but includes additional steps to enable legacy cipher suites. We used the following configuration command:

```
./configure \
    --prefix=/usr/local/nginx2 \
    --with-cc-opt='-g -O2 -fno-omit-frame-pointer \
        -mno-omit-leaf-frame-pointer \
        -ffile-prefix-map=/build/nginx-DlMnQR/nginx-1.27.3=. \
        -flto=auto -ffat-lto-objects -fstack-protector-strong \
        -fstack-clash-protection -Wformat -Werror=format-security \
        -fcf-protection \
        -fdebug-prefix-map=/build/nginx-DlMnQR/nginx-1.27.3=/usr/src/nginx-1.27.3 \
        -fPIC -Wdate-time ' \
    --with-ld-opt='-Wl,-Bsymbolic-functions -flto=auto -ffat-lto-objects \
        -Wl,-z,relro -Wl,-z,now -fPIC' \
    --conf-path=/etc/nginx2/nginx.conf \
    --error-log-path=/var/log/nginx2/error.log \
    --http-log-path=/var/log/nginx2/access.log \
    --pid-path=/var/run/nginx2.pid \
    --lock-path=/var/lock/nginx2.lock \
    --modules-path=/usr/local/nginx2/modules \
    --http-client-body-temp-path=/var/lib/nginx2/body \
    --http-fastcgi-temp-path=/var/lib/nginx2/fastcgi \
    --http-proxy-temp-path=/var/lib/nginx2/proxy \
    --http-scgi-temp-path=/var/lib/nginx2/scgi \
    --http-uwsgi-temp-path=/var/lib/nginx2/uwsgi \
    --with-compat \
    --with-debug \
    --with-pcre-jit \
    --with-http_ssl_module \
    --with-http_stub_status_module \
    --with-http_realip_module \
    --with-http_auth_request_module \
    --with-http_v2_module \
    --with-http_dav_module \
    --with-http_slice_module \
    --with-threads \
    --with-http_addition_module \
    --with-http_flv_module \
    --with-http_gunzip_module \
    --with-http_gzip_static_module \
    --with-http_mp4_module \
    --with-http_random_index_module \
    --with-http_secure_link_module \
    --with-http_sub_module \
    --with-mail_ssl_module \
    --with-stream_ssl_module \
    --with-stream_ssl_preread_module \
    --with-stream_realip_module \
    --with-http_geoip_module=dynamic \
    --with-http_image_filter_module=dynamic \
    --with-http_perl_module=dynamic \
    --with-http_xslt_module=dynamic \
    --with-mail=dynamic \
    --with-stream=dynamic \
    --with-stream_geoip_module=dynamic \
    --with-openssl=/home/okan/openssl-1.0.2u
```

Before running `make`, we modified the `Makefile` in the `objs` directory to ensure the legacy OpenSSL features are included:

```
/home/okan/openssl-1.0.2u/.openssl/include/openssl/ssl.h: objs/Makefile
cd /home/okan/openssl-1.0.2u \
&& if [ -f Makefile ]; then $(MAKE) clean; fi \
&& ./config --prefix=/home/okan/openssl-1.0.2u/.openssl \
    no-shared no-threads zlib \
    enable-weak-ssl-ciphers enable-ssl2 enable-rc5 enable-rc2 \
    enable-cms enable-md2 enable-mdc2 enable-ec enable-ec2m \
    enable-ecdh enable-ecdsa enable-seed enable-camellia enable-idea \
    enable-rfc3779 \
&& $(MAKE) \
&& $(MAKE) install_sw LIBDIR=lib
```

The modified `Makefile` ensures support for legacy ciphers such as RC4, MD5, and export-grade ciphers. Afterward, we installed the instance using:

```
make
sudo make install
```

### 5.1.3 Flask Integration

We used Flask for responding to HTTPS requests and generating dynamic iframe responses. We performed the following steps to set up Flask:

1. Install Python virtual environment support:

```
sudo apt install python3.8-venv
```

2. Create and activate a virtual environment:

```
python3 -m venv ~/flask_env
source ~/flask_env/bin/activate
```

3. Install Flask and required libraries:

```
pip install flask cryptography
```

The Flask application generates unique iframe paths for each request and logs connection details, including the TLS version and cipher suite.

## 5.2 Client Setup

We ran the client on a virtual machine (VM) with Ubuntu 20.04.6 LTS installed.

We installed OpenVPN 2.4.12 on the client using the following command:

```
sudo apt install openvpn
```

The first instance of OpenSSL, version 1.1.1f, is preinstalled on Ubuntu 20.04. Alternatively, it can be installed with:

```
sudo apt install openssl
```

We configured the second instance of OpenSSL, version 1.0.2u, with the following command:

```
./config --prefix=/usr/local/openssl1.0.2 \
         --openssldir=/usr/local/openssl1.0.2 \
         shared zlib enable-weak-ssl-ciphers enable-ssl2 enable-rc5 \
         enable-rc2 enable-GOST enable-cms enable-md2 enable-mdc2 \
         enable-ec enable-ec2m enable-ecdh enable-ecdsa enable-seed \
         enable-camellia enable-idea enable-rfc3779 -DOPENSSL_USE_BUILD_DATE
```

This configuration ensures that as many cipher suites as possible are activated. It is important to note that we needed to run the following command before installing:

```
make depend
```

After that, we built and installed the OpenSSL version using:

```
make
sudo make install
```

## 5.3 Server Implementation

The server setup involves two instances of Nginx, each using a different version of OpenSSL, to simulate a variety of TLS configurations. This section describes the configuration and interaction between these two instances, as well as the role of Flask for request handling.

### 5.3.1 First Nginx Instance (Modern Cipher Suites)

This Nginx instance is responsible to handle the more modern cipher suites. It uses OpenSSL 1.1.1f, which in comparison to OpenSSL 1.0.2u also supports TLS 1.3.

We configured this Nginx instance to use the ngx stream module because this server is the first access point for the internet in our test setup. For this reason, we set up the server as a TCP passthrough. The configuration for that looks like this:

```
server {
    listen 443;
    proxy_pass $upstream_server;
    ssl_preread on;
    proxy_protocol on;
}
```

The server listens on the HTTPS port 443 and passes the packet through depending on the SNI. For that, `ssl-preread` is needed and used so that SNI can be read without establishing a TLS connection already. Also, we need the proxy protocol to ensure that the servers will get accurate client IP and port information instead of the IP and port of the Nginx instance.

We used the following configuration for the stream:

```
map $ssl_preread_server_name $upstream_server {
    default         legacy_openssl;
    www.proxy2.com  modern_openssl;
    proxy2.com      modern_openssl;
}
```

This sets the legacy server as default because it might lure more observers/attackers to the server if they see that older cipher suites are supported. We chose the modern server when the SNI is set to `(www.)proxy2.com`.

In this instance of Nginx, the HTTPS server listens on port 8002. It is important that the proxy protocol is enabled. We configured it as follows:

```
listen 8002 ssl proxy_protocol;
```

With that, the server is capable of receiving the client IP from the stream server:

```
set_real_ip_from 127.0.0.1;
real_ip_header proxy_protocol;
```

We configured the server to support TLS 1.0 - TLS 1.3. We defined the following for the cipher configurations:

```
ssl_ciphers ALL:@SECLEVEL=0;
```

This ensures that the server does not reject ciphers that could potentially be unsafe. This configuration allows the server to accept and negotiate a wide range of cipher suites, including weaker ones that are required for the experiment.

Incoming requests are forwarded to the Flask application. Additionally, connection information such as the host, real IP, protocol, and other details are sent to the Flask application using the following configuration:

```
location / {
    proxy_pass http://127.0.0.1:5000; # Forward to Flask app
    proxy_set_header SSL-PROTOCOL $ssl_protocol;
    proxy_set_header SSL-CIPHER $ssl_cipher;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}
```

We configured the server to use two certificates: one RSA certificate and one DSA (Digital Signature Algorithm) certificate. Additionally, we included Diffie-Hellman (DH) parameters in the configuration to support as many cipher suites as possible.

We created the RSA certificate with the following command:

```
sudo /usr/bin/openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ./private/nginx-selfsigned.key -out
↪  ./certs/nginx-selfsigned.crt
```

This command generates an RSA private key with a length of 2048 bits and a self-signed certificate valid for 365 days. We left all certificate information at the default values, except for the CN (Common Name), which we set to `proxy2.com`.

We created the DSA certificate with the following commands:

```
/usr/bin/openssl dsaparam -out dsaparam.pem 2048
/usr/bin/openssl gendsa -out server-dss.key dsaparam.pem
/usr/bin/openssl req -new -x509 -key server-dss.key -out server-dss.crt -days 365
```

We generated the DSA private key with a key length of 2048 bits. Like the RSA certificate, the DSA certificate is self-signed and valid for 365 days.

We generated the Diffie-Hellman (DH) parameters with the following command:

```
/usr/bin/openssl dhparam -out dhparam.pem 1024
```

We chose a key length of 1024 bits for the DH parameters, which is considered weak by modern cryptographic standards.

### 5.3.2 Second Nginx Instance (Legacy Cipher Suites)

We built this Nginx instance with OpenSSL 1.0.2u to support export-grade and other weak cipher suites.

In this instance of Nginx, the HTTPS server listens on port 8001. It is crucial that the proxy protocol is enabled, similar to the first Nginx instance. In general, the configuration of this server is very similar to the first one. We set the server name to `www.proxy1.com` and `proxy1.com`. This server also uses the proxy protocol and `real_ip_header`, ensuring that the real client IP is forwarded correctly. To allow the widest possible range of cipher suites, the server is configured to accept all ciphers, including `eNULL`. This enables the detection of potential observers attempting to retransmit a unique request without encryption. This server also uses two certificates and Diffie-Hellman (DH) parameters to support a broad range of legacy cipher suites.

We created the RSA certificate with the following command:

```
sudo /usr/bin/openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ./private/nginx-selfsigned.key -out
↪  ./certs/nginx-selfsigned.crt
```

This command generates an RSA private key with a length of 2048 bits and a self-signed certificate valid for 365 days. We kept all certificate details at their default values, except for the Common Name (CN), which we set to `proxy1.com`. We created the DSA certificate with the following commands:

```
/usr/local/openssl1.0.2/bin/openssl dsaparam -out dsaparam.pem 1024
/usr/local/openssl1.0.2/bin/openssl gendsa -out server-dss.key dsaparam.pem
/usr/local/openssl1.0.2/bin/openssl req -new -x509 -key server-dss.key -out server-dss.crt -days 365
```

We chose a key length of 1024 bits to ensure compatibility with legacy cipher suites.

We generated the Diffie-Hellman (DH) parameters with the following command:

```
/usr/local/openssl1.0.2/bin/openssl dhparam -out dhparam.pem 512
```

We chose a key length of 512 bits to support older cipher suites, especially the export-grade cipher suites. However, this is considered insecure by modern cryptographic standards. The reduced key length increases the likelihood that an attacker could compute the shared secret using modern computational resources, but for the purpose of this experiment, it enables testing of legacy cryptographic mechanisms.

Identical to the first Nginx instance, this server also forwards incoming HTTPS requests to the Flask application for processing. For this, we utilized the same Flask application used in the first instance. This setup ensures that we correctly forward all relevant metadata about the connection, including the selected cipher suite and protocol version, to the Flask application. The Flask application processes the incoming requests and logs the necessary information for subsequent analysis.

### 5.3.3 Flask Application

The Flask application handles all requests for both Nginx instances. To achieve this, the application uses dynamic routing. As described in the methodology, we define the unique paths we track as follows:

```
@app.route('/path/<encrypted_path_hex>', methods=['GET'])
```

We designed the Flask application to attempt decryption of the `encrypted_path_hex` using AES-GCM. If decryption is successful, the application logs the decrypted path, which contains information about the timestamp, TLS version, and cipher suite used in the connection. Additionally, Flask logs the client IP, virtual host (vHost), User-Agent, TLS version, and cipher suite as received and processed by the Nginx server.

Furthermore, the application generates a timestamp and logs both the encrypted path and the corresponding iframe path. We constructed the iframe path as follows:

```
inside_path = f"https://{server-ip}/c/{encrypt(encrypted_path_hex,key)}"
```

This means the iframe path is an encrypted version of the original encrypted path. The Flask application responds to such requests with a response containing only the iframe.

We handle iframe paths in a similar manner. However, in addition to logging the iframe path, we also record the corresponding main path. We log all other metadata, including the client IP, TLS version, and cipher suite, similarly to the main path.

The response to an iframe request is a simple HTTP 200 status. For any other requests that do not match the predefined paths, the application returns an HTTP 404 ”Page Not Found” error.

## 5.4 Client Implementation

The client is responsible for initiating TLS connections to the server using various cipher suites and TLS versions. Its primary objectives are:

- Establishing TLS connections using both modern and legacy OpenSSL versions.
- Sending unique, encrypted requests containing metadata about the connection (timestamp, TLS version, and cipher suite).
- Logging connection results to identify successful and failed cipher suites.
- Testing connections over different VPN paths to detect possible TLS interceptions.

We implemented the client as a Python script running on an Ubuntu 20.04.6 LTS virtual machine. It utilizes two OpenSSL versions:

- **Modern OpenSSL (1.1.1f)** for TLS 1.3 and strong cipher suites.
- **Legacy OpenSSL (1.0.2u)** for weak and export-grade ciphers no longer supported in modern OpenSSL.

### 5.4.1 Generating and Encrypting Unique Paths

To track and detect potential TLS interceptions, the client sends unique paths in its requests. We encrypt these paths using AES-GCM before including them in the HTTP request.

#### 5.4.1.1 Key Generation and Encryption

We derived a static encryption key using PBKDF2 with the following parameters:

- **Password:** `"tls_testing_password"`
- **Salt:** `"tls_testing_salt"`
- **Key Length:** 128-bit (16 bytes)
- **Iterations:** 100,000

Each request contains an encrypted path that includes:

- `time=` Timestamp of the request.
- `tls=` TLS version used for the connection.
- `cipher=` Cipher suite used for the connection.

Example plaintext metadata before encryption:

```
time=2025-01-29T22:49:52.049270&tls=tls1_1&cipher=ADH-DES-CBC-SHA
```

After encryption, we convert the data to hexadecimal format and include it in the request URL:

```
https://proxy1.com/path/<AES-GCM-encrypted-hex>
```

### 5.4.2 Establishing TLS Connections with OpenSSL

The client establishes TLS connections using OpenSSL's `s_client` command. The general structure of the OpenSSL command is:

```
openssl s_client -connect <host:port> <tls-version> -servername <sni> -cipher <cipher> -ign_eof
```

In Python, we dynamically built this command as follows:

```
command = [
    openssl_path, "s_client",
    "-connect", f"{host}:{port}",
    f"-{tls_version}",
    "-servername", servername,
    "-ign_eof"
]
if not is_tls13:
    command.extend(["-cipher", cipher])
```

The `-servername` option ensures that the client uses the correct Server Name Indication (SNI). The `-cipher` option applies only to TLS versions prior to TLS 1.3, as the installed OpenSSL instance does not allow manual selection of cipher suites in TLS 1.3.

### 5.4.3 Sending the HTTP Request

After establishing the TLS connection, the client sends an HTTP request that includes the unique encrypted path

```
request = f"GET {path} HTTP/1.1\r\nHost: {servername}\r\nConnection: keep-alive\r\n\r\n"
process.stdin.write(request)
process.stdin.flush()
```

The client then reads the server's response to check if the connection was successful:

- **200 OK:** The request was processed successfully.
- **HTTP error code:** The request failed.
- **No response:** The TLS handshake failed.

### 5.4.4 Logging Connection Results

The script stores results in the following files:

- `connection_results.txt` – Contains all connection attempts.
- `successful_ciphers.txt` – Lists only successful cipher suites.
- `failed_ciphers.txt` – Lists cipher suites that failed to establish a connection.
- `openssl_commands.txt` – Logs the executed OpenSSL commands for debugging.

### 5.4.5 Testing Cipher Suites Across Different TLS Versions

The client tests all available cipher suites for both OpenSSL instances across multiple TLS versions. It uses the modern OpenSSL instance (1.1.1f) for TLS 1.3, TLS 1.2, and TLS 1.1, while the legacy OpenSSL instance (1.0.2u) enables testing of cipher suites that modern OpenSSL no longer supports, including export ciphers, DES, RC4, and other weak ciphers.

Instead of filtering specific cipher suites, the client retrieves the full list of supported ciphers from each OpenSSL version and systematically attempts connections using every cipher with the corresponding TLS versions.

To obtain the list of supported ciphers, the client executes the following command for each OpenSSL instance:

```python
def get_ciphers(openssl_path):
    result = subprocess.run([openssl_path, "ciphers", "-v", "ALL"],
                            capture_output=True, text=True)
    ciphers = []
    if result.returncode == 0:
        for line in result.stdout.splitlines():
            cipher = line.split()[0]
            ciphers.append(cipher)
    return ciphers
```

For each cipher, the client attempts connections using the appropriate TLS versions:

- Modern OpenSSL (1.1.1f) tests cipher suites with `TLS 1.3`, `TLS 1.2`, and `TLS 1.1`.
- Legacy OpenSSL (1.0.2u) tests cipher suites with `TLS 1.2`, `TLS 1.1`, and `TLS 1.0`.

Additionally, we manually added the `eNULL` cipher suite to the list of supported cipher suites, as OpenSSL does not include it by default.

This ensures that the client tests all potential cipher suite and TLS version combinations.

**Table 5.1:** Selection of Tested Cipher Suites

| TLS Version | Cipher Suite |
|---|---|
| **Strong and Modern Cipher Suites** | |
| TLS 1.3 | TLS_AES_256_GCM_SHA384 |
| TLS 1.2 | ECDHE-RSA-AES256-GCM-SHA384 |
| TLS 1.2 | ECDHE-RSA-CHACHA20-POLY1305 |
| TLS 1.2 | AES256-GCM-SHA384 |
| TLS 1.2 | CAMELLIA256-SHA256 |
| **Weak and Deprecated Cipher Suites** | |
| TLS 1.1 | IDEA-CBC-SHA |
| TLS 1.0 | ECDHE-RSA-RC4-SHA |
| TLS 1.0 | ADH-RC4-MD5 |
| TLS 1.0 | DES-CBC3-SHA |
| TLS 1.0 | EXP-EDH-RSA-DES-CBC-SHA |
| TLS 1.0 | eNULL (No Encryption) |

The table provides an overview of selected cipher suites tested during the experiment. A complete list of all successfully tested cipher suites, categorized by TLS version, can be found in Appendix A.8.

### 5.4.6 Testing with OpenVPN

The client utilizes a script to connect to selected NordVPN configurations. These configurations route only the connection to the destination server through the VPN tunnel, leaving other traffic unaffected. This ensures that the client's true IP address is masked, simulating connections from various geographical locations.

After establishing the VPN connection, the client runs the Python script to initiate TLS connections using all available cipher suites. We repeat this process for each selected NordVPN configuration to ensure that the request script is executed from multiple geographical locations, increasing the likelihood of encountering potential interception attempts.

We selected the following NordVPN server configurations for the experiment:

**Table 5.2:** Selected NordVPN Configurations Sorted by Continent

| NordVPN Configuration | Country | Censorship/Interception |
|---|---|---|
| | **North America** | |
| ca1635.nordvpn.com.tcp.ovpn | Canada | No |
| mx54.nordvpn.com.tcp.ovpn | Mexico | No |
| us9811.nordvpn.com.tcp.ovpn | United States | No |
| | **South America** | |
| ar61.nordvpn.com.tcp.ovpn | Argentina | No |
| br75.nordvpn.com.tcp.ovpn | Brazil | No |
| co6.nordvpn.com.tcp.ovpn | Colombia | No |
| | **Europe** | |
| al53.nordvpn.com.tcp.ovpn | Albania | No |
| de1153.nordvpn.com.tcp.ovpn | Germany | No |
| is76.nordvpn.com.tcp.ovpn | Iceland | No |
| is78.nordvpn.com.tcp.ovpn | Iceland | No |
| it215.nordvpn.com.tcp.ovpn | Italy | No |
| pt103.nordvpn.com.tcp.ovpn | Portugal | No |
| rs76.nordvpn.com.tcp.ovpn | Serbia | No |
| uk1836.nordvpn.com.tcp.ovpn | United Kingdom | No |
| | **Asia** | |
| ae55.nordvpn.com.tcp.ovpn | United Arab Emirates | Yes [25] |
| il58.nordvpn.com.tcp.ovpn | Israel | No |
| jp744.nordvpn.com.tcp.ovpn | Japan | No |
| kr114.nordvpn.com.tcp.ovpn | South Korea | No |
| sg546.nordvpn.com.tcp.ovpn | Singapore | Yes [16] |
| tr54.nordvpn.com.tcp.ovpn | Turkey | Yes [17] |
| tw193.nordvpn.com.tcp.ovpn | Taiwan | No |
| | **Africa** | |
| ng5.nordvpn.com.tcp.ovpn | Nigeria | Yes [15] |
| za128.nordvpn.com.tcp.ovpn | South Africa | No |
| | **Oceania** | |
| au649.nordvpn.com.tcp.ovpn | Australia | No |
| nz98.nordvpn.com.tcp.ovpn | New Zealand | No |

These configurations represent a diverse range of geographical locations and network environments, enhancing the experiment's ability to detect potential interception attempts from various vantage points.

# 6 Evaluation and Discussion

## 6.1 Data Analysis

We analyzed the data collected from the server logs—including IP addresses, access paths, TLS versions, and cipher suites—to identify patterns indicative of TLS interception. The analysis focuses on detecting retransmissions of unique requests, discrepancies between the intended and the actual TLS configurations, and unauthorized access to iframe paths. Notably, a total of 22,758 connections were established from various VPN endpoints during the experiment, originating from 144 distinct IP addresses.

## 6.2 Key Findings

Our analysis of the server logs revealed that:

- We observed no retransmissions of unique requests.
- There was no evidence of unauthorized access to iframe paths.
- The cipher suites and TLS versions embedded within the unique request paths match the TLS configurations reported by the Nginx server.

## 6.3 Interpretation

Our analysis of the server logs revealed no retransmissions of unique requests and no unauthorized iframe accesses. This suggests that, within the scope of this experiment, we observed no clear evidence of active TLS interception. However, it is important to note that this result does not definitively prove that no attacker is present. The detection method employed in this experiment relies on the assumption that an attacker, if intercepting TLS connections, would retransmit unique requests or access the corresponding iframe paths. If an attacker chooses not to interact with these unique paths or there is nothing indicative of an interception in the logged request, an interception will not be detected.

Moreover, even if we do not find any discrepancies between the TLS configuration embedded in the unique request and the actual TLS parameters observed at the server, this does not guarantee that interception attempts are completely absent. Instead, our observations indicate that, for the tested connections and under the given network conditions, we did not detect any manipulation of the handshake or request paths. Therefore, while our experiment provides valuable insights into the behavior of TLS connections under various configurations, it cannot conclusively rule out the possibility of TLS interception in all scenarios.

## 6.4 Additional Connection Statistics

We conducted the experiment on the following dates: 2025-01-22, 2025-01-27, 2025-01-29, 2025-01-30, 2025-01-31, 2025-02-01, 2025-02-05, 2025-02-06, 2025-02-09, 2025-02-11, 2025-02-12, 2025-02-13, 2025-02-14, and 2025-02-15.

We deliberately spread our connection tests over these dates to capture diverse network conditions. Notably, no connections or handshakes were unexpectedly interrupted, except for a few instances due to a server shutdown. While we focused on the 94 cipher suites that established successful connections A.8, our script also attempted combinations that we expected to fail due to limitations in OpenSSL or certificate prerequisites. We intentionally excluded these failed attempts from further analysis, as our primary interest was in the reliably functioning connections.

## 6.5 Limitations

Our experiment has several limitations:

- **Duration of the Experiment:** Detecting TLS interception requires a significant number of connection attempts. A longer observation period could increase the likelihood of detecting interception.
- **Limited Network Diversity:** We were limited by the number of available VPN servers, which reduced the diversity of network paths tested.
- **Detection Method Assumptions:** The approach assumes that an attacker will either retransmit a unique request or access the iframe path. However, more sophisticated interception methods might evade detection. Also, if an attacker is capable of decrypting the messages between the client and server, they may recognize that the main path only returns the iframe path, which could deter them from accessing it. Therefore, the absence of iframe accesses does not conclusively rule out the presence of an attacker.
- **Lack of Valuable Content:** The server currently does not serve any content other than the paths. This may reduce the attractiveness of the server to potential attackers, as there is little incentive to intercept or manipulate the traffic.

## 6.6 Implications and Future Directions

For future work, the experimental setup could be enhanced in several ways:

- **Longer and More Frequent Testing:** Extending the duration of the experiment and increasing the frequency of requests would likely improve the detection rate of TLS interceptions.
- **Expanded VPN Configurations:** Utilizing a greater number of VPN servers would simulate a broader array of network conditions, potentially increasing the likelihood of detecting interception attempts.
- **Enhanced Honeypot Content:** To make the server a more attractive target for attackers, future experiments could simulate a scenario in which the server appears to host valuable information, such as an administrative panel, sensitive credentials, or a password manager. This could increase the chances that an attacker would attempt to access the paths.

In summary, while the experiment did not produce definitive evidence of TLS interceptions under the tested conditions, its design provides a valuable framework for further investigation. The current results indicate that no observable manipulation of the TLS handshake or paths occurred during the test period. However, due to the experiment's limitations and its reliance on specific attacker behaviors, the absence of such evidence should be interpreted with caution. Future work addressing the above points may yield a more comprehensive understanding of TLS interception in real-world scenarios.

# 7 Future Work

## 7.1 ServerHello Fingerprinting

In this section, we introduce a method for detecting TLS interceptions using ServerHello fingerprinting.

### 7.1.1 Attacker Model

We assume an attacker performing a Man-in-the-Middle (MitM) attack. The attacker sits on-path between the client and the server, intercepting the communication and establishing separate TLS connections with both. The attacker impersonates the server when communicating with the client and impersonates the client when communicating with the server. This positioning enables the attacker to manipulate the TLS handshakes with both parties and intercept or manipulate the data exchanged between them.

### 7.1.2 ServerHello Fingerprinting

For fingerprinting, we capture the network packet from the client side and extract the following information from the ServerHello message:

- TLS version
- Cipher suite
- Extensions

We then construct a fingerprint input string by concatenating the hexadecimal representations of these values, separated by colons.

```
fingerprint_input = f"{hex(version)}:{hex(cipher)}:{','.join([hex(e) for e in extensions])}"
```

Finally, we compute the SHA-256 hash of this string to generate a unique fingerprint:

```
fingerprint = hashlib.sha256(fingerprint_input.encode('utf-8')).hexdigest()
```

This fingerprint can be used to identify inconsistencies or deviations from expected behavior in the ServerHello message, which could indicate the presence of an attacker. The full code for this fingerprinting method is provided in Appendix A.7. Please note that this code is specifically designed for packets captured on the tuno interface, which lacks an Ethernet layer. This is because the code was developed and tested in a virtual machine environment where network traffic is routed through a VPN tunnel.

To make this method more reliable, it is important to know how the server will construct the ServerHello message. This method can be integrated with the methodology described in Section 4 because the client knows which TLS version, cipher suite, and extensions will be used for each connection, regardless of the VPN connection.

For this method to be more reliable, it is important to ensure that the ServerHello message has some degree of uniqueness, such as in the extensions and their order. One approach to achieve this would be to add custom extensions to the ServerHello message. This would make it more difficult for an attacker to forge a ServerHello message that perfectly matches the legitimate server's response, as the attacker would need to replicate the custom extensions as well.

Server fingerprinting is only effective if the construction of the ServerHello message is predictable. This technique can be easily integrated with the unique request method described in Section 4 because the expected structure of the ServerHello messages is known beforehand. However, applying server fingerprinting becomes challenging when the server is not under our control or when the composition of the ServerHello message is not precisely predictable.

Furthermore, this method assumes that the attacker does not mimic the server's ServerHello message, including the TLS version, cipher suite, and extensions (including any custom extensions). Therefore, it is necessary to introduce an element that the attacker cannot replicate. One possibility is to use a custom extension to transmit information encrypted with a key shared with the client. For example, the server could include a custom extension in the ServerHello message containing the client's IP address, port, and the timestamp of the Client Hello message, all encrypted with a key known to the client. Since the attacker establishes a separate connection to the server, it is expected that their connection will have a different timestamp in the TCP layer compared to the client's original connection. The client can then compare this information with its own Client Hello message to detect discrepancies.

This approach is feasible only if the VPN does not alter the TCP layer or if the VPN servers are under our control, allowing us to manipulate the relevant values.

### 7.1.3 Considerations and Limitations

This detection method requires prior knowledge of the expected ServerHello message structure to identify anomalies and deviations. It also relies on the assumption that an attacker will not meticulously analyze and replicate the original connection messages, including the TLS version, cipher suite, and extensions in the correct order. While techniques like embedding encrypted timestamps in custom extensions can be employed to mitigate this, they introduce dependencies on other methods.

However, this method offers the advantage of potentially detecting a strong attacker capable of performing on-path attacks and impersonating both the server and client. Another advantage is that if an attacker does not develop their own tool for sending ServerHello messages or copies the ServerHello information (including custom extensions), standard libraries are unlikely to send the custom extensions from the server's ServerHello message.

In summary, ServerHello fingerprinting can be a valuable technique for detecting TLS interception, particularly when combined with other methods and when the server configuration is known or predictable. However, it is essential to be aware of its limitations, especially when dealing with sophisticated attackers who might attempt to forge ServerHello messages.

# 8 | Conclusion

In this thesis, we investigated methods for detecting TLS interception in real-world scenarios by implementing a honeypot server and a client that systematically tested various TLS configurations and cipher suites. We simulated connections from different geographical locations using VPNs to increase the likelihood of encountering interception attempts.

To achieve this, we designed an automated client that iterated through all available cipher suites for different OpenSSL versions and established TLS connections with our honeypot server. The client embedded details about the intended TLS configuration within the request path, allowing us to later verify whether the observed parameters at the server matched the expected values. By utilizing multiple VPN endpoints spread across various countries, we aimed to test TLS interception across diverse network conditions. Additionally, our server logged all incoming requests, including client IP addresses, TLS parameters, and accessed paths, providing a comprehensive dataset for post-experiment analysis.

Our analysis of the server logs revealed no clear evidence of active TLS interception within the scope of our experiment. We did not observe any retransmissions of unique requests or unauthorized iframe accesses. Additionally, the TLS configurations reported by the server consistently matched the intended configurations embedded in the unique requests.

However, it is important to acknowledge the limitations of this experiment. The duration and frequency of testing, the diversity of VPN configurations, and the assumptions made about attacker behavior may have influenced the results. Additionally, the lack of valuable content on the server might have reduced its attractiveness as a target for attackers.

Despite these limitations, this thesis provides a valuable framework for future research on TLS interception detection. Further studies could address the limitations of this experiment by extending the testing duration, expanding the range of VPN configurations, and enhancing the honeypot content to attract more attackers.

In conclusion, while our experiment did not produce definitive evidence of TLS interception, it contributes to a better understanding of TLS interception techniques and offers insights for developing more robust detection methods. The findings highlight the importance of continuous monitoring and testing of TLS connections to ensure secure communication in the ever-evolving threat landscape.

# References

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Oct. 2015. URL: https://doi.org/10.1145/2810103.2813707.

[2] A. Arampatzis. *Why It's Dangerous to Use Outdated TLS Security Protocols*. Accessed on November 14, 2024. July 2020. URL: https://venafi.com/blog/why-its-dangerous-use-outdated-tls-security-protocols/.

[3] A. Arampatzis. *What Are SSL Stripping Attacks?* Accessed on January 27, 2025. Dec. 2023. URL: https://venafi.com/blog/what-are-ssl-stripping-attacks/.

[4] H. Böck, J. Somorovsky, and C. Young. "Return of Bleichenbacher's Oracle Threat (ROBOT)". In: *Proceedings of the 27th USENIX Security Symposium*. Aug. 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bock.

[5] A. Brodie. *Overview of TLS v1.3*. OWASP Chapter London Presentation Slides. Jan. 2018. URL: https://owasp.org/www-chapter-london/assets/slides/OWASPLondon20180125_TLSv1.3_Andy_Brodie.pdf.

[6] D. Chetlall. *A New Way of Detecting TLS (SSL) MITM Attacks*. Accessed on January 25, 2025. Aug. 2022. URL: https://www.enea.com/insights/a-new-way-of-detecting-tls-ssl-mitm-attacks/.

[7] J. Clark and P. Oorschot. "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements". In: May 2013. DOI: 10.1109/SP.2013.41.

[8] Cloudflare. *Transport Layer Security (TLS)*. Accessed on November 14, 2024. URL: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/.

[9] Cloudflare. *What happens in a TLS handshake? | SSL handshake*. Accessed on January 25, 2025. URL: https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/.

[10] J. Curguz. "Vulnerabilities of the SSL/TLS protocol". In: *Computer Science and Information Technology*. May 2016. DOI: 10.5121/csit.2016.60620.

[11] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Internet Standard. Aug. 2008. URL: https://datatracker.ietf.org/doc/html/rfc5246.

[12] L. Encrypt. *Let's Encrypt Frequently Asked Questions*. Accessed on March 3, 2025. URL: https://letsencrypt.org/docs/faq/.

[13] Faddom. *The Evolution From SSL to TLS*. Accessed on March 3, 2025. Aug. 2023. URL: https://faddom.com/ssl-to-tls/.

[14] G. Fisher and L. Valenta. *Monsters in the Middleboxes: Introducing Two New Tools for Detecting HTTPS Interception*. Accessed on January 31, 2025. Mar. 2019. URL: https://blog.cloudflare.com/monsters-in-the-middleboxes/.

[15] Freedom House. *Nigeria: Freedom on the Net 2024*. Accessed on March 3, 2025. 2024. URL: https://freedomhouse.org/country/nigeria/freedom-net/2024.

[16] Freedom House. *Singapore: Freedom on the Net 2024*. Accessed on March 3, 2025. 2024. URL: https://freedomhouse.org/country/singapore/freedom-net/2024.

[17] Freedom House. *Turkey: Freedom on the Net 2024*. Accessed on March 3, 2025. 2024. URL: https://freedomhouse.org/country/turkey/freedom-net/2024.

[18] A. S. Gillis. *VPN (Virtual Private Network)*. Accessed on February 2, 2025. URL: https://www.computerweekly.com/de/definition/Virtuelles-Privates-Netzwerk-Virtual-Private-Network-VPN.

[19] M. Green. *Attack of the week: FREAK (or 'factoring the NSA for fun and profit')*. Accessed on January 25, 2025. Mar. 2015. URL: https://blog.cryptographyengineering.com/2015/03/03/attack-of-week-freak-or-factoring-nsa/.

[20] Radware. *ARP Poisoning*. Accessed on January 25, 2025. URL: https://www.radware.com/security/ddos-knowledge-center/ddospedia/arp-poisoning/.

[21] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Internet Standard. Aug. 2018. URL: https://datatracker.ietf.org/doc/html/rfc8446.

[22] A. Vigderman and G. Turner. *Internet Censorship*. Accessed on March 4, 2025. Aug. 2024. URL: https://www.security.org/vpn/internet-censorship/.

[23] Virtue Security. *Export Ciphers Enabled*. Accessed on January 25, 2025. URL: https://www.virtuesecurity.com/kb/export-ciphers-enabled/.

[24] W3Techs. *Usage statistics of HTTPS for websites*. Accessed on February 2, 2025. Feb. 2025. URL: https://w3techs.com/technologies/breakdown/ce-httpsdefault/ranking.

[25] A. Wilkens. *VAE drohen mit Gefängnisstrafe für Majestätsbeleidigung im Web*. Accessed on March 3, 2025. Nov. 2012. URL: https://www.heise.de/news/VAE-drohen-mit-Gefaengnisstrafe-fuer-Majestaetsbeleidigung-im-Web-1749565.html.

[26] Y. Xing, C. Lu, B. Liu, H. Duan, J. Sun, and Z. Li. "Yesterday Once More: Global Measurement of Internet Traffic Shadowing Behaviors". In: *Proceedings of the 2024 ACM Internet Measurement Conference (IMC 24)*. Nov. 2024. DOI: 10.1145/3646547.3689023.

[27] Zbigniew Banach. *How the BEAST Attack Works: Reading Encrypted Data Without Decryption*. Accessed on January 25, 2025. Nov. 2024. URL: https://www.invicti.com/blog/web-security/how-the-beast-attack-works/.

# A | Appendix

## A.1 Client Code

```python
import subprocess
import argparse
import os
from datetime import datetime
import hashlib
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend

tested_combinations = set()

results = []

# AES-128 Key
# Generate a static key using PBKDF2
password = b"tls_testing_password"
salt = b'tls_testing_salt'
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=16,
    salt=salt,
    iterations=100000,
    backend=default_backend()
)
key = kdf.derive(password)

# Encrypt using AES-GCM
def encrypt(data, key):
    aesgcm = AESGCM(key)
    nonce = os.urandom(12)
    ciphertext = aesgcm.encrypt(nonce, data.encode(), None)
    return nonce + ciphertext

# Get all Cipher from openssl
def get_ciphers(openssl_path):
    result = subprocess.run([openssl_path, "ciphers", "-v", "ALL"], capture_output=True, text=True)
    ciphers =
    if result.returncode == 0:
        for line in result.stdout.splitlines():
            cipher = line.split()
            ciphers.append(cipher)
    return ciphers

# Function to test a connection with a specific cipher and TLS version
def test_connection(openssl_path, cipher, tls_version, host, port, ignore_cert, servername, is_tls13=False):
    # Set environment variable for keylogging if necessary ()
    # if openssl_path == args.legacy_openssl:
    #     os.environ["SSLKEYLOGFILE"] = args.keylog_file

    # Create path with timestamp, TLS version, and cipher
    path_elements = f"time={datetime.now().isoformat()}&tls={tls_version}&cipher={cipher}"
    encrypted_path_hex = encrypt(path_elements, key).hex()
    path = f"/path/{encrypted_path_hex}"

    request = f"GET {path} HTTP/1.1\r\nHost: {servername}\r\nConnection: keep-alive\r\n\r\n"
    command = [
        openssl_path, "s_client", "-connect", f"{host}:{port}", f"-{tls_version}", "-servername", servername,
        ↪  "-ign_eof"
    ]
    if not is_tls13:
        command.extend(["-cipher", cipher])
    if ignore_cert:
       # openssl ignores certs validation per default
    if openssl_path == args.modern_openssl:
        command.extend(["-keylogfile", args.keylog_file])

    try:
        process = subprocess.Popen(
            command,
            stdin=subprocess.PIPE,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True
        )
        # Send the HTTP request
        stdout_lines =
        process.stdin.write(request)
        process.stdin.flush()

        while True:
            line = process.stdout.readline()
            if line == "" and process.poll() is not None:
                break
            stdout_lines.append(line)
            if "200 OK" in line or "HTTP" in line:
                # Close the connection cleanly
                process.stdin.write("close\n")
                process.stdin.flush()
                process.terminate()
                break

        stdout = "".join(stdout_lines)

        # Look for HTTP 200 OK in the response
        if "200 OK" in stdout:
            result = f"SUCCESS WITH PATH: TLS={tls_version}, Cipher={cipher}, Path={path},
            ↪  Path_Elements={path_elements}"
        elif "HTTP" in stdout:
            result = f"SUCCESS WITHOUT PATH: TLS={tls_version}, Cipher={cipher}, Path={path},
            ↪  Path_Elements={path_elements}"
        else:
            result = f"FAILED: TLS={tls_version}, Cipher={cipher}, Path={path},
            ↪  Path_Elements={path_elements}, Response={stdout.strip()}"
    except Exception as e:
        process.kill()
        result = f"FAILED: TLS={tls_version}, Cipher={cipher}, Path={path}, Path_Elements={path_elements},
        ↪  Error: {e}"

    print(result)
    with open("openssl_commands.txt", "a") as f:
        f.write(" ".join(command) + "\n")
    return result

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Test TLS connections with different configurations.")
    parser.add_argument("host", help="The host to connect to.")
    parser.add_argument("port", help="The port to connect to.")
    parser.add_argument("--modern_openssl", default="/usr/bin/openssl", help="Path to the modern OpenSSL
    ↪  version.")
    parser.add_argument("--legacy_openssl", default="/usr/local/openssl1.0.2/bin/openssl", help="Path to the
    ↪  legacy OpenSSL version.")
    parser.add_argument("--keylog_file", default="/home/okan/keylog.txt", help="File to store TLS keys.")
    parser.add_argument("--ignore_cert", action="store_true", help="Ignore certificate errors.") # openssl
    ↪  doesnt care as default
    args = parser.parse_args()

    # Test modern OpenSSL with TLS 1.3
    print("Testing modern OpenSSL with TLS 1.3...")
    result = test_connection(args.modern_openssl, "TLS_AES_256_GCM_SHA384", "tls1_3", args.host, args.port,
    ↪  args.ignore_cert, "proxy2.com", is_tls13=True)
    results.append(result)

    # Test modern OpenSSL with other TLS versions
    print("Testing modern OpenSSL with other TLS versions...")
    modern_ciphers = get_ciphers(args.modern_openssl)
    for cipher in modern_ciphers:
        for tls_version in ["tls1_2", "tls1_1"]:
            if (tls_version, cipher) not in tested_combinations:
                result = test_connection(args.modern_openssl, cipher, tls_version, args.host, args.port,
                ↪  args.ignore_cert, "proxy2.com")
                results.append(result)
                tested_combinations.add((tls_version, cipher))

    # Test legacy OpenSSL
    print("Testing legacy OpenSSL...")
    legacy_ciphers = get_ciphers(args.legacy_openssl)
    if "eNULL" not in legacy_ciphers:
        legacy_ciphers.append("eNULL")

    for cipher in legacy_ciphers:
        for tls_version in ["tls1", "tls1_1", "tls1_2"]:
            if (tls_version, cipher) not in tested_combinations:
                result = test_connection(args.legacy_openssl, cipher, tls_version, args.host, args.port,
                ↪  args.ignore_cert, "proxy1.com")
                results.append(result)
                tested_combinations.add((tls_version, cipher))

    # Write results to files
    with open("connection_results.txt", "w") as f:
        for result in results:
            f.write(result + "\n")

    with open("successful_ciphers.txt", "w") as f:
        for result in results:
            if "SUCCESS" in result:
                tls_version = result.split("TLS=").split(",")
                cipher = result.split("Cipher=").split(",")
                f.write(f"{tls_version} - {cipher}\n")

    with open("failed_ciphers.txt", "w") as f:
        for result in results:
            if "FAILED" in result:
                tls_version = result.split("TLS=").split(",")
                cipher = result.split("Cipher=").split(",")
                f.write(f"{tls_version} - {cipher}\n")

    print("Successful ciphers saved to successful_ciphers.txt")
    print("Failed ciphers saved to failed_ciphers.txt")
    print("Results saved to connection_results.txt")
    print(f"TLS keys saved to {args.keylog_file}")
    print("OpenSSL commands saved to openssl_commands.txt")
```

## A.2 VPN Connection and Script Execution Code

```bash
#!/bin/bash

TARGET_IP="185.73.23.156"
TARGET_PORT="443"

VPN_CONFIG_DIR="/etc/openvpn/ovpn_tcp/"

PYTHON_SCRIPT="/home/okan/connectingTLS.py"

run_python_script() {
  echo "start connectingTLS.py with IP: $TARGET_IP and Port: $TARGET_PORT"
  sudo python3 "$PYTHON_SCRIPT" "$TARGET_IP" "$TARGET_PORT"

  if [ $? -ne 0 ]; then
    echo "Error while running connectingTLS.py"
    exit 1
  fi
}

wait_for() {
  local seconds=$1
  echo "wait $seconds sec"
  sleep "$seconds"
}

if [ "$EUID" -ne 0 ]; then
  echo "has to be used with sudo"
  exit 1
fi

if [! -f "$PYTHON_SCRIPT" ]; then
  echo "error: python-script '$PYTHON_SCRIPT' nicht gefunden."
  exit 1
fi

# first run without vpn
run_python_script

find "$VPN_CONFIG_DIR" -maxdepth 1 -name "*.ovpn" -print0 | while IFS= read -r -d $'\0' vpn_config; do

  vpn_config_name=$(basename "$vpn_config")

  echo "----------------------------------------"
  echo "using VPN-config: $vpn_config_name"

  echo "start openvpn with $vpn_config_name"
```

```
    sudo openvpn --config "$vpn_config" --daemon
    if [ $? -ne 0 ]; then
        echo "error starting openvpn: $vpn_config_name"
        exit 1
    fi

    # wait to make sure tunnel is active
    wait_for 15

    run_python_script

    echo "stop OpenVPN"
    sudo pkill -f "openvpn"

    # Wait to make sure openvpn stopped
    wait_for 7

    echo "----------------------------------------"
done

echo "script finished."
exit 0
```

## A.3 First Nginx Instance Configuration

```
worker_processes  1;
load_module modules/ngx_http_geoip_module.so;
load_module modules/ngx_stream_module.so;
load_module modules/ngx_stream_geoip_module.so;

events {
    worker_connections  1024;
}

stream {
    # Map for SNI-detection to choose the Upstream
    map $ssl_preread_server_name $upstream_server {
        default          legacy_openssl;       # Standard-Backend
        www.proxy2.com modern_openssl;    # Legacy-Backend für alte OpenSSL
        proxy2.com modern_openssl;
    }

    # Upstreams definieren
    upstream modern_openssl {
        server 127.0.0.1:8002; # Backend für moderne OpenSSL
    }

    upstream legacy_openssl {
        server 127.0.0.1:8001; # Backend für Legacy OpenSSL
    }

    # TLS-Passthrough-Server
    server {
        listen 443;
        proxy_pass $upstream_server; # Dynamische Auswahl des Backends
        ssl_preread on; # Erlaubt SNI-Erkennung vor der TLS-Verhandlung
        proxy_protocol on; # proxy protocol aktivieren

    }

    # Logging for Stream

    log_format custom_stream '$remote_addr:$remote_port -> $server_addr:$server_port '
                              'via $upstream_addr - SNI: $ssl_preread_server_name';
    access_log /var/log/nginx/stream_access.log custom_stream;
    error_log /var/log/nginx/stream_error.log;

}


http {
    include       mime.types;
    default_type  application/octet-stream;
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log warn;
    sendfile        on;
    keepalive_timeout  65;

    # HTTPS server
    server {

         listen 8002 ssl proxy_protocol;
         listen [::]:8002 ssl proxy_protocol;

         server_name  www.proxy2.com proxy2.com;

         set_real_ip_from 127.0.0.1;
         real_ip_header proxy_protocol;

         ssl_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;
         ssl_prefer_server_ciphers on;
         ssl_ciphers ALL:@SECLEVEL=0;
         ssl_certificate /home/okan/nginx/certs/nginx-selfsigned.crt;
         ssl_certificate_key /home/okan/nginx/private/nginx-selfsigned.key;
                # DSA-Zertifikat und Schlüssel
         ssl_certificate /home/okan/nginx/certs/server-dss.crt;
         ssl_certificate_key /home/okan/nginx/private/server-dss.key;
                # DH-Parameter für ADH-Ciphers
         ssl_dhparam /home/okan/nginx/dhparam/dhparam.pem;
         location / {
                 proxy_pass http://127.0.0.1:5000; # Forward to Flask app
                 proxy_set_header SSL-PROTOCOL $ssl_protocol;
                 proxy_set_header SSL-CIPHER $ssl_cipher;
                 proxy_set_header Host $host;
                 proxy_set_header X-Real-IP $remote_addr;
                 proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                 proxy_set_header X-Forwarded-Proto $scheme;
         }
    }

}
```

## A.4 Second Nginx Instance Configuration

```

worker_processes  1;
load_module modules/ngx_http_geoip_module.so;
load_module modules/ngx_stream_module.so;
load_module modules/ngx_stream_geoip_module.so;
events {
    worker_connections  1024;
}
```

```

http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
    # HTTPS server

    server {
        listen       8001 ssl proxy_protocol;
        server_name  www.proxy1.com proxy1.com;

        set_real_ip_from 127.0.0.1;
        real_ip_header proxy_protocol;

         # RSA-Zertifikat und Schlüssel
        ssl_certificate /home/okan/nginx2/certs/nginx-selfsigned.crt;
        ssl_certificate_key /home/okan/nginx2/private/nginx-selfsigned.key;

        # DSA-Zertifikat und Schlüssel
        ssl_certificate /home/okan/nginx2/certs/server-dss.crt;
        ssl_certificate_key /home/okan/nginx2/private/server-dss.key;

        # DH-Parameter für ADH-Ciphers
        ssl_dhparam /home/okan/nginx2/dhparam/dhparam.pem;

        ssl_session_cache    shared:SSL:1m;
        ssl_session_timeout  5m;

        ssl_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;
        ssl_ciphers         ALL:eNULL;
        ssl_prefer_server_ciphers on;

        location / {

            proxy_pass http://127.0.0.1:5000; # Forward to Flask app
            proxy_set_header SSL-PROTOCOL $ssl_protocol;
            proxy_set_header SSL-CIPHER $ssl_cipher;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }
    }
}
```

## A.5 Flask Application Code

```python
from flask import Flask, request, render_template_string, render_template
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend
from datetime import datetime
import binascii
import os


app = Flask(__name__)

password = b"tls_testing_password"
```

```python
salt = b'tls_testing_salt'
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=16,
    salt=salt,
    iterations=100000,
    backend=default_backend()
)
key = kdf.derive(password)

def decrypt(encrypted_data, key):
    """Decrypts AES-GCM encrypted data with a given key"""
    data = binascii.unhexlify(encrypted_data)
    nonce = data[:12]  # Extract the nonce
    ciphertext = data[12:] # Extract the ciphertext
    aesgcm = AESGCM(key)
    try:
        decrypted_data = aesgcm.decrypt(nonce, ciphertext, None)
        return decrypted_data.decode()  # Decode assuming it was a UTF-8 string
    except Exception as e:
        print(f"Decryption failed: {e}")
        return None

def encrypt(data, key):
    """Encrypts data using AES-GCM with a given key"""
    aesgcm = AESGCM(key)
    nonce = os.urandom(12)
    ciphertext = aesgcm.encrypt(nonce, data.encode(), None)
    return binascii.hexlify(nonce + ciphertext).decode()

@app.route('/path/<encrypted_path_hex>', methods=['GET'])
def dynamic_path(encrypted_path_hex):
    decrypted_path_elements = decrypt(encrypted_path_hex, key)

    # Zusätzliche Informationen
    client_ip = request.remote_addr

    user_agent = request.headers.get('User-Agent', 'Unknown')
    tls_version = request.headers.get('SSL-PROTOCOL', 'Unknown')  # TLS-Version aus Header
    cipher_suite = request.headers.get('SSL-CIPHER', 'Unknown')  # Cipher Suite aus Header
    vHost= request.headers.get('Host', 'Unknown')
    x_real_ip=request.headers.get('X-Real-IP', 'Unknown')
    x_forwarded_for=request.headers.get('X-Forwarded-For', 'Unknown')

    if decrypted_path_elements:

         # generate iframe path
        inside_path = f"https://185.73.23.156/c/{encrypt(encrypted_path_hex,key)}"

        # logging acces details
        with open('access_log.txt', 'a') as log:
            log.write(
                f"Accessed path: {decrypted_path_elements}\n"
                f"Client IP: {client_ip}\n"
                f"x_real_ip: {x_real_ip}\n"
                f"x_forwarded_for: {x_forwarded_for}\n"
                f"Host: {vHost}\n"
                f"User-Agent: {user_agent}\n"
                f"TLS Version: {tls_version}\n"
                f"Cipher Suite: {cipher_suite}\n"
                f"Timestamp: {datetime.utcnow().isoformat()}\n"
                f"Encrpted Path: {encrypted_path_hex}\n"
                f"Inside Path: {inside_path}\n"
                f"{'-'*60}\n"
            )

        response_content = f"""
        <!DOCTYPE html>
        <html>
        <head>
            <title>Valid Path</title>
        </head>
        <body>
            <h1>Valid path accessed: </h1>
            <iframe src="{inside_path}" style="width: 100%; height: 300px; border: none;"></iframe>


        </body>
        </html>
        """
        return render_template_string(response_content), 200
    else:
        return "Invalid request: Page not found.", 404

@app.route('/c/<encrypted_data>', methods=['GET'])
def inside_path(encrypted_data):
    decrypted_path = decrypt(encrypted_data, key)

    # Zusätzliche Informationen
    client_ip = request.remote_addr
    user_agent = request.headers.get('User-Agent', 'Unknown')
    tls_version = request.headers.get('SSL-PROTOCOL', 'Unknown')  # TLS-Version aus Header
    cipher_suite = request.headers.get('SSL-CIPHER', 'Unknown')  # Cipher Suite aus Header
    vHost = request.headers.get('Host', 'Unknown')
    x_real_ip=request.headers.get('X-Real-IP', 'Unknown')
    x_forwarded_for=request.headers.get('X-Forwarded-For', 'Unknown')

    if decrypted_path:
        # logging access details
        with open('access_log_inside.txt', 'a') as log:
            log.write(
                f"Accessed Main path: {decrypt(encrypted_data,key)}\n"
                f"Main path Elements: {decrypt(decrypt(encrypted_data,key),key)}\n"
                f"Client IP: {client_ip}\n"
                f"x_real_ip: {x_real_ip}\n"
                f"x_forwarded_for: {x_forwarded_for}\n"
                f"Host: {vHost}\n"
                f"User-Agent: {user_agent}\n"
                f"TLS Version: {tls_version}\n"
                f"Cipher Suite: {cipher_suite}\n"
                f"Timestamp: {datetime.utcnow().isoformat()}\n"
                f"Inside Path Value: {encrypted_data}\n"
                f"{'-'*60}\n"
            )
        return f"This is the inside path for: ", 200
    else:
        return "Invalid request: Page not found.", 404
#@app.route('/debug')
#def debug():
#     return {
#         "remote_addr": request.remote_addr,
#         "x_real_ip": request.headers.get('X-Real-IP'),
#         "x_forwarded_for": request.headers.get('X-Forwarded-For')
#     }
```

```python

# Fallback route for invalid paths
@app.errorhandler(404)
def page_not_found(e):
    return "Invalid request: Page not found.", 404

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

## A.6 Log analysis

```bash
    #!/bin/bash

access_path=""

while IFS= read -r line; do


  if [[ $line == *"Accessed path:"* ]]; then
    access_path=$line
  fi


  if [[ $line == *"Cipher Suite:"* ]]; then
    cipher_suite=$(echo "$line" | awk -F"Cipher Suite: " '{print $2}')


    accessed_cipher=$(echo "$access_path" | grep -oP '(?<=cipher=)[^&]+')


    if [[ "$accessed_cipher" != "$cipher_suite" ]] && ! [[ "$accessed_cipher" == "eNULL" && "$cipher_suite"
    ↪  == "ECDHE-RSA-NULL-SHA" ]]; then
      echo "--------------------------------------------------------------"
      echo "$access_path"
      echo "$line"
      echo "Cipher im Accessed Path: $accessed_cipher"
      echo "Cipher Suite: $cipher_suite"
      echo "--------------------------------------------------------------"
    fi

    # Access Path zurücksetzen
    access_path=""
  fi

done < access_log.txt
```

```bash
#!/bin/bash


access_path=""

while IFS= read -r line; do


  if [[ $line == *"Accessed path:"* ]]; then
    access_path=$line
  fi


  if [[ $line == *"TLS Version:"* ]]; then

```

```
    tls_version=$(echo "$line" | grep -oP '(?<=TLSv)\d+(\.\d+)?' | tr -d '[:space:]' | tr -d '\r')


    accessed_tls=$(echo "$access_path" | grep -oP '(?<=tls=tls)\d+(_\d+)?' | tr '_' '.' | tr -d '[:space:]' | tr -d '\r')

    if [[ "$accessed_tls" != "$tls_version" ]]; then
      echo "------------------------------------------------------------"
      echo "$access_path"
      echo "$line"
      echo "TLS im Accessed Path: $accessed_tls"
      echo "TLS Version: $tls_version"
      echo "------------------------------------------------------------"
    fi

    access_path=""
  fi

done < access_log.txt
```

## A.7 Server Hello fingerprinting

```python
import dpkt
import sys
import struct
import hashlib

def parse_tls_server_hello(data):
    if len(data) < 5:
        return None

    content_type, version, length = struct.unpack('!BHH', data[:5])
    if content_type != 22:  # TLS Handshake
        return None

    handshake_type = data[5]
    if handshake_type != 2:  # Server Hello
        return None

    pos = 43  # Skip handshake type, length ...
    session_id_length = data[pos]

    pos += 1 + session_id_length  # Skip session ID

    if pos + 2 > len(data):
        return None

    selected_cipher_suite = struct.unpack('!H', data[pos:pos+2])[0]

    pos += 2
    compression_method = data[pos]
    pos += 1

    if pos + 2 > len(data):
        return None

    extensions_length = struct.unpack('!H', data[pos:pos+2])[0]
    pos += 2

    extpos = pos

    extensions = []

    while pos + 4 <= extpos + extensions_length:
        ext_type, ext_len = struct.unpack('!HH', data[pos:pos+4])
        pos += 4 + ext_len
        extensions.append(ext_type)

    return version, selected_cipher_suite, extensions
def process_pcap(file_path):
    fingerprints = []
    with open(file_path, 'rb') as f:
        pcap = dpkt.pcap.Reader(f)
        for _, buf in pcap:
            try:
                ip = dpkt.ip.IP(buf)
            except Exception as e:
                print(f"Skipping malformed packet: {e}")
                continue
            #print(ip.data)
            if not isinstance(ip.data, dpkt.tcp.TCP):
                print("Skipping non-TCP packet")
                continue

            tcp = ip.data

            if len(tcp.data) > 0:
                print(f"Processing packet with length: {len(tcp.data)} bytes")
                fingerprint = parse_tls_server_hello(tcp.data)
                if fingerprint:
                    print(f"Found TLS Server Hello: Version={hex(fingerprint[0])}, Cipher={hex(fingerprint[1])}, Extensions={[hex(e) for e in fingerprint[2]]}")
                    fingerprints.append(fingerprint)

    with open("Fingerprints.txt", "w") as output_file:
        for version, cipher, extensions in fingerprints:
            fingerprint_input = f"{hex(version)}:{hex(cipher)}:{','.join([hex(e) for e in extensions])}"
            fingerprint = hashlib.sha256(fingerprint_input.encode('utf-8')).hexdigest()
            output_file.write(f"TLS Version: {hex(version)}, Cipher: {hex(cipher)}, Extensions: {[hex(e) for e in extensions]} Fingerprint:{fingerprint}\n")
    print("Results written to Fingerprints.txt")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python script.py <pcap_file>")
        sys.exit(1)
    process_pcap(sys.argv[1])
```

## A.8 Complete List of Tested Cipher Suites

**Table A.1:** Successfully Tested Cipher Suites

| TLS Version | Cipher Suite |
|---|---|
| TLS 1.3 | TLS_AES_256_GCM_SHA384 |
| TLS 1.2 | ECDHE-RSA-AES256-GCM-SHA384 |
| TLS 1.2 | ECDHE-RSA-CHACHA20-POLY1305 |
| TLS 1.2 | ECDHE-ARIA256-GCM-SHA384 |
| TLS 1.2 | ECDHE-RSA-AES128-GCM-SHA256 |
| TLS 1.2 | ECDHE-ARIA128-GCM-SHA256 |
| TLS 1.2 | ECDHE-RSA-AES256-SHA384 |
| TLS 1.2 | ECDHE-RSA-CAMELLIA256-SHA384 |
| TLS 1.2 | ECDHE-RSA-AES128-SHA256 |
| TLS 1.2 | ECDHE-RSA-CAMELLIA128-SHA256 |
| TLS 1.2 | ECDHE-RSA-AES256-SHA |
| TLS 1.2 | ECDHE-RSA-AES128-SHA |
| TLS 1.2 | AES256-GCM-SHA384 |
| TLS 1.2 | AES256-CCM8 |
| TLS 1.2 | AES256-CCM |
| TLS 1.2 | ARIA256-GCM-SHA384 |
| TLS 1.2 | AES128-GCM-SHA256 |
| TLS 1.2 | AES128-CCM8 |
| TLS 1.2 | AES128-CCM |
| TLS 1.2 | ARIA128-GCM-SHA256 |
| TLS 1.2 | AES256-SHA256 |
| TLS 1.2 | CAMELLIA256-SHA256 |
| TLS 1.2 | AES128-SHA256 |
| TLS 1.2 | CAMELLIA128-SHA256 |
| TLS 1.2 | AES256-SHA |
| TLS 1.2 | CAMELLIA256-SHA |
| TLS 1.2 | AES128-SHA |
| TLS 1.2 | SEED-SHA |
| TLS 1.2 | CAMELLIA128-SHA |
| TLS 1.0, 1.1, 1.2 | ECDHE-RSA-RC4-SHA |
| TLS 1.0, 1.1, 1.2 | AECDH-RC4-SHA |
| TLS 1.0, 1.1, 1.2 | ADH-RC4-MD5 |
| TLS 1.0, 1.1, 1.2 | RC4-SHA |
| TLS 1.0, 1.1, 1.2 | RC4-MD5 |
| TLS 1.0, 1.1, 1.2 | ECDHE-RSA-DES-CBC3-SHA |
| TLS 1.0, 1.1, 1.2 | AECDH-DES-CBC3-SHA |
| TLS 1.0, 1.1, 1.2 | ADH-DES-CBC3-SHA |
| TLS 1.0, 1.1, 1.2 | DES-CBC3-SHA |
| TLS 1.0, 1.1, 1.2 | ADH-DES-CBC-SHA |
| TLS 1.0, 1.1, 1.2 | DES-CBC-SHA |
| TLS 1.0, 1.1, 1.2 | EXP-EDH-RSA-DES-CBC-SHA |
| TLS 1.0, 1.1, 1.2 | EXP-EDH-DSS-DES-CBC-SHA |
| TLS 1.0, 1.1, 1.2 | EXP-ADH-DES-CBC-SHA |
| TLS 1.0, 1.1, 1.2 | EXP-ADH-RC4-MD5 |
| TLS 1.0, 1.1, 1.2 | eNULL |
| TLS 1.0 | ECDHE-RSA-AES256-SHA |
| TLS 1.0 | AECDH-AES256-SHA |
| TLS 1.0 | ADH-AES256-SHA |
| TLS 1.0 | ADH-CAMELLIA256-SHA |
| TLS 1.0 | AES256-SHA |
| TLS 1.0 | CAMELLIA256-SHA |
| TLS 1.0 | ECDHE-RSA-AES128-SHA |
| TLS 1.0 | AECDH-AES128-SHA |
| TLS 1.0 | ADH-AES128-SHA |
| TLS 1.0 | ADH-SEED-SHA |
| TLS 1.0 | ADH-CAMELLIA128-SHA |
| TLS 1.0 | AES128-SHA |
| TLS 1.0 | SEED-SHA |
| TLS 1.0 | CAMELLIA128-SHA |
| TLS 1.0, 1.1, 1.2 | IDEA-CBC-SHA |