SNIT: A Modified TLS Handshake Protocol for Censorship Circumvention

by

© *Yixuan Liang*

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Engineering

Department of Electrical and Computer Engineering

Faculty of Engineering and Applied Science

Memorial University of Newfoundland

October 2024

St. John's Newfoundland

Abstract

Internet censorship is a global problem. Many countries censor the internet for different reasons. This threatens internet freedom and access to information. 82.8% of websites use the Transport Layer Security (TLS) protocol, which significantly enhances security. However, weaknesses exposed by TLS can still be exploited for internet censorship. For example, the unencrypted Server Name Indication (SNI) directly reveals the website's identity. We propose a modified handshake protocol, SNIT, for both TLS 1.2 and TLS 1.3, making it difficult to conduct SNI-based censorship. SNIT has high resistance to active probing. On average, the performance loss is 31.69 ms per TLS connection, and there is no effect on subsequent traffic. Compared to competitive approaches, SNIT has decent overall security and performance.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Jonathan Anderson, for his continuous support, guidance, and encouragement throughout my research journey. His insightful feedback and expertise have been invaluable in shaping this dissertation. Without his mentorship, this work would not have been possible.

I am profoundly grateful to my parents for their unwavering love, support, and understanding. Their sacrifices and encouragement have been the foundation upon which I have built my academic endeavors. Their belief in my potential has been a constant source of motivation.

I also wish to extend my sincere thanks to Mitacs, C-CORE and the Department of National Defence for providing the financial support necessary to conduct this research. Their generous funding made it possible to access essential resources and dedicate my time to this project.

Finally, I would like to thank my friends and colleagues for their companionship and support during this journey. Their encouragement and camaraderie have been invaluable.

Thank you all for being an integral part of this accomplishment.

Contents

List of Figures

List of Tables

List of symbols, Nomenclature or Abbreviations

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

CDN Content Delivery Network

CN Commom Name

DF Domain Fronting

DDoS Distributed Denial-of-service Attack

DH Diffie-Hellman

DHE Diffie-Hellman Ephemeral

DNS Domain Name System

DoH Domain Name System over Hypertext Transfer Protocol Secure

DoS Denial of Service Attack

DSS Digital Signature Algorithm

ECC Elliptic Curve Cryptography

ECDSA Elliptic Curve Digital Signature Algorithm

ECH Encrypted ClientHello

ESNI Encrypted Server Name Indication

GCM Galois/Counter Mode

GFW Great Firewall

HMAC Hash-based Message Authentication Code

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IP Internet Protocol

MAC Message Authentication Code

PRF Pseudorandom Function

RSA Rivest–Shamir–Adleman

RST Reset

SHA Secure Hash Algorithm

SNI Server Name Indication

SNIT Server Name Indication Tunneling

TLS Transport Layer Security

Tor The Onion Router

UDP User Datagram Protocol

VPS Virtual Private Server

Chapter 1

Introduction

1.1 Internet censorship overview

The internet has become indispensable in people's lives. We can get a tremendous amount of information from the internet, some of which is perceived as harmful by governments. Censorship is the selective suppression of sensitive and offensive content [4]. Governments censor internet resources, usually websites or specific content on a website, and control what people can access. Internet censorship can be divided into technical censorship and non-technical censorship. Non-technical censorship refers to administrative and legal means: publishing or distributing sensitive content risks arrest. On the contrary, technical censorship refers to the selective blocking of internet resources using technical method by governments, institutions and companies. Compared to government censorship, institutions and companies usually have more effective ways than network-based blocking, e.g. device monitoring. In this thesis, we focus on government censorship.

One of the most famous censorship projects is China's Great Firewall (GFW) [3]. People living in mainland China are subject to strict internet censorship. Other countries in the world, from Iran [5], Turkey [6], and India [7] to France [8] also have different levels of internet censorship. Internet censorship restricts people's right to access information freely, and different countries have very different levels and topics of censorship. Table 1.1 shows the different topics censored by five countries. All sensitive topics are censored in China, especially those related to politics. Political censorship also happens in Saudi Arabia [9] and Syria [9]. They are also relatively strict in religion. Cuba is as strict as China in politics, especially corruption [3]. France has only had censorship since the terrorist attacks of 2015.

Internet censorship is based on screening: blocking sensitive content while releasing others. This target is impossible to achieve perfectly: over and under-blocking always happens. If users could encrypt and obfuscate everything perfectly, and the censor could never identify any website being accessed by the user, it would have to make a choice: blocking or allowing everything.

1.2 Technical blocking methods

In this section, the technical methods of blocking websites will be briefly introduced. As new protocols improve privacy, additional blocking methods are also applied to compete with them. The methods below are listed chronologically.

Table 1.1: Censored topics by country [3]

Topic censored/ Countries	China	Saudi Arabia	Syria	Cuba	France
Criticism of authorities	XX	XX	XX	XX	
Corruption	XX	X		XX	
Conflict	XX	XX	XX		X
Political Opposition	XX	XX	XX	XX	
Mobilization for public causes	XX	XX	XX	XX	
Social commentary	XX	X		X	
Blasphemy	X	XX			
Satire	XX				
LGBTI issues	X	XX			
Ethnic and religious minority	XX	XX	XX		

Note: an "XX" indicates severe censorship, and an "X" indicates limited censorship.

1.2.1 HTTP keyword filtering

In this obsolete method, hostnames in unencrypted HTTP requests or contents on specific web pages are compared with a watchlist. If they are on the watchlist, the firewall will disguise itself as both sides to send TCP RST packets to the opposite side, and the connection will be terminated. With TLS, HTTP traffic is encrypted, and no keywords can be intercepted. Currently, 82.8% of websites use HTTPS as the default protocol [10], so this method has little effect, especially against websites serving sensitive content.

1.2.2 Domain Name System (DNS) manipulation

DNS translates a domain name into one or more IP addresses. Before accessing a website, the client usually needs to query a DNS server and get the corresponding IP address. DNS query packets are not encrypted by default, meaning that the firewall can check unencrypted DNS query packets. If the website in the query is banned, the firewall will send the user a response containing fake IP addresses. This method can be bypassed by DNS over HTTPS (DoH) [11], which encrypts the DNS query packets with HTTPS. However, methods targeting HTTPS, i.e., all methods described in this section except HTTP keyword filtering, can also be applied against the DoH servers. In China, 99% of DoH queries are dropped [12]. If all DNS servers supporting DoH are blocked (or at least all servers known to the user), the user has to use traditional DNS and is vulnerable. DoH responses may also be manipulated if the DoH server is deployed in the regions that suffer DNS manipulation [13]. If an upper-level DNS server is being manipulated, it may pass the fake IP addresses to the lower-level DoH

servers.

1.2.3 Internet Protocol (IP) address blocking

In this method, the firewall records the IP addresses belonging to the blacklisted websites. All packets sent to these IP addresses, or packets belonging to specific protocols (e.g. HTTP and HTTPS), will be discarded. This method has two significant disadvantages. First, if the target website and multiple other websites are hosted on the target IP, they will be blocked together. This is especially serious for content delivery networks (CDN). Second, most websites change their IP addresses frequently. Strict IP blocking does not work well and leads to many overblocks. Although IP blocking is applied in most countries, it is always a secondary means [14, 15].

1.2.4 Certificate filtering

For TLS versions earlier than 1.2, the server certificate is always sent without encryption. The common name (CN) is an indispensable part of a digital certificate. It must be the same as the website name, and the firewall can detect it [16]. In TLS 1.3, the server certificate is encrypted, so this method has no effect.

1.2.5 Server Name Indication (SNI)-based censorship

A server name indication (SNI) is sent in plaintext in all TLS versions. Like the common name in the digital certificate, it is the name of the website that the client is accessing. If the server is hosting multiple websites, SNI allows the server to pick the corresponding certificate. For example, a CDN may host a large number

of domains. The censor can identify the server with SNI and perform SNI-based censorship [17, 18, 19]. This is the main issue addressed in this research and will be introduced in more detail in Chapter 3.

1.3 Circumvention tools and countermeasures

Many tools have been developed to fight against internet censorship. Most of them appear in the form of bridges. The bridges are deployed outside the firewall and provide an alternative path between the users and the blocked websites. All network traffic, or at least all the traffic to the blocked websites, must be forwarded by the servers. This comes at a cost. As a result, either the users pay for these tools or the quality of service cannot be guaranteed. These tools can also be blocked; the cat-and-mouse game has been going on for years. A brief introduction to these tools will be presented here.

Figure 1.1: A Tor circuit with three relay servers between the user and google.com

The Onion Router (Tor) [20] is a widely used tool for censorship circumvention.

It provides strong security and anonymity. Under default settings, three Tor servers are between the user and the website, as shown in Figure 1.1. The network traffic is encrypted and routed three times. In the figure, the packets from the user are sent to 54.36.205.38 first. Then, they will be redirected to 91.121.219.14 and 185.220.102.248 before they reach the google.com server. Neither the website nor the servers in Germany or Belgium know the user's identity, including the IP address. Tor is funded by the US government, private foundations, and individual donors [21]. Since Tor is free for users, its bandwidth is limited to several Mbps. Also, the long journey raises the latency to hundreds of milliseconds. Moreover, its security faces severe tests. China's Great Firewall has been blocking Tor for years [22, 23], though Tor's developers have developed new tools to fight against it [24]. Many studies impact its anonymity. In [25, 26], the attacker can perform deanonymization in honey relays. The relay server can be replaced and controlled by the attacker. In [27], the bitcoin addresses have been proven to be exploitable, and the Tor users might be linked to the hidden services.

Since public servers are targeted by censors, tools based on Virtual Private Server (VPS) become widespread. A VPS is a virtual machine that provides internet hosting services. Users have access to the operating system, and multiple VPSs can be run on a physical server. The user can build a private network with a VPS. One popular solution is V2Ray + TLS + websocket + web [28]. V2Ray is a proxy tool that supports multiple inbound and outbound protocols. The core idea of this solution is to hide the encrypted proxy behind a website. The website is only an empty shell built by the server and can be accessed with TLS. The VPS forwards encrypted network traffic while the censor thinks the user is just accessing the website. V2Ray may still

get blocked, but the method of identifying it is still uncertain. Some related studies are based on traffic analysis [28, 29, 30].

1.4 Goal for this work

Although many security protocols and censorship circumvention tools have been applied, most of them are either out-of-date or limited. There is no perfect solution. In this research, our goal is to develop a practical and deployable circumvention solution for SNI-based censorship. It should take into account security, convenience and performance at the same time.

Chapter 2

Background

2.1 Transport Layer Security overview

Transport Layer Security (TLS) is a security protocol. It provides privacy, data integrity and authentication between two communicating applications [31]. It works on top of a transport layer protocol, usually TCP. It can also be used with UDP [32]. The protocol is widely used in network applications, including Hypertext Transfer Protocol Secure (HTTPS). HTTP is only designed for data communication and does not provide confidentiality, data integrity, or authentication. If the attacker intercepts the packets, all communications between the client and the server will be received. More seriously, the attacker can disguise themselves as the server. In the worst-case scenario described in Dolev-Yao model [33], the user communicates only with the attacker and only receives messages from the attacker. In this situation, HTTP cannot provide confidentiality or data integrity. HTTPS is HTTP encrypted with TLS. Currently, 82.8% of websites use HTTPS as the default protocol [10]. After

many upgrades, the latest version is TLS 1.3 [34]. The most widely-used version is still TLS 1.2, which is supported by 99.9% of popular websites [35].

2.2 TLS handshake overview

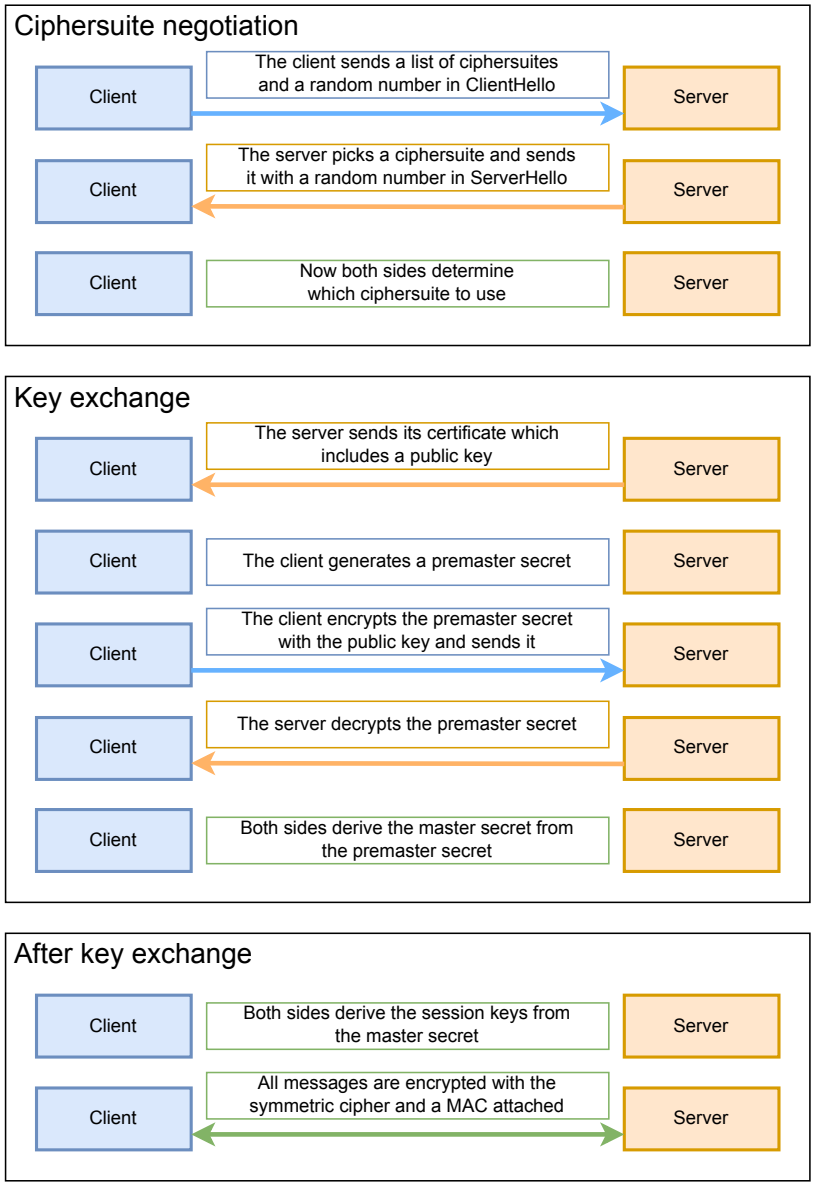

Figure 2.1: Summary of the TLS handshake

Cryptographic algorithms provide data confidentiality and integrity in TLS. The cryptographic algorithms are described in the cipher suite. In TLS 1.2, a cipher suite includes a key exchange algorithm, an authentication method, a symmetric cipher, and a hash algorithm. An example of a TLS 1.2 cipher suite name is TLS_DHE_RSA_WITH_AES_256_GCM_SHA384: Ephemeral Diffie-Hellman (DHE) is the key exchange algorithm, Rivest–Shamir–Adleman (RSA) is the authentication mechanism, AES-256-GCM is the symmetric cipher, in which 256 is the key size, and Galois/Counter Mode (GCM) is the block cipher mode. Secure Hash Algorithm (SHA)-384 is the hash algorithm.

Figure 2.1 summarizes a TLS handshake. A cipher suite is first negotiated: the client provides a list of supported cipher suites in its ClientHello message, and the server picks one of them in the ServerHello message. Then, the key exchange is performed. The purpose of the key exchange is to generate a master secret on both sides, a 48-byte secret that is used to derive session keys for the symmetric cipher. The key exchange is secure over a public channel, though both sides have no prior knowledge of each other. Even if the attacker intercepts the handshake messages, it still cannot get the key. After the key exchange, all messages are encrypted by the symmetric cipher. A message authentication code (MAC) is also attached to each message to ensure data integrity.

In a key exchange, first, the client generates a random number and combines it with the latest TLS version the client supports. This is called the premaster secret. The premaster secret is then encrypted using the public key from the server's certificate. The client sends the encrypted premaster secret to the server, and the server decrypts it with the certificate's private key. Both sides now share the same

premaster secret. The master secret is derived from the premaster secret and the ClientHello and ServerHello messages by the formula in Eqn. 2.1 [31]:

$$\texttt{master\_secret} = PRF(\texttt{pre\_master\_secret}, \text{“}\texttt{master\_secret}\text{”}, \texttt{ClientHello.random} + \texttt{ServerHello.random}) \tag{2.1}$$

PRF (pseudorandom function) is the hash algorithm in the cipher suite. The ClientHello and ServerHello random numbers are used to prevent replay attacks. In a replay attack [36], the attacker intercepts the packets in a session and re-transmits them. The server will discard a ClientHello message if its random number has previously been seen.

A symmetric cipher is also specified in the negotiated cipher suite. Its keys and other parameters are derived from the master secret. The attacker can only decrypt a message if they acquires the master secret or at least a session key or breaks the symmetric cipher.

Data integrity for each message is protected by a message authentication code (MAC). Each message must have the correct MAC, or the connection will be aborted. For block cipher modes which do not provide authenticated encryption with associated data (AEAD), a hash-based message authentication code (HMAC) is used. AEADs have built-in MAC-like functionality, so the HMAC is not needed: HMAC and AEAD share a similar working principle. As an example, Eqn. 2.2 calculates the HMAC [37]:

$$HMAC(K, m) = H((K' \oplus opad)||H(K' \oplus ipad)||m) \tag{2.2}$$

where H is a cryptographic hash function, and m is the message. The hash function must have at least a weak form of collision resistance [38]. k' is the secret key if the

key is smaller than the block size of the hash function. Otherwise, it is the hash value of the key. The opad (outer pad) and ipad (inner pad) are fixed values. They make the secret keys pseudorandom. One can only calculate the HMAC of a message with the secret key. The attacker cannot replace a message with a new one and generate the correct HMAC. This structure can also prevent other attacks [39, 40], and it has no known flaws.

2.3 Digital certificate

In a handshake, the server must provide a digital certificate for authentication. The client will verify its validity. The client will proceed to the next step if the certificate is valid. Otherwise, the client will receive an alert that indicates an attacker might be impersonating the server. The server can also require the client to provide a certificate to verify its identity.

To confirm the validity of a certificate, the following parts must be verified: the common name (CN) of the issuer and the subject, the validity period, and the digital signature. The issuer's CN in each certificate must be identical to the subject's CN in the upper-level certificate. For example, if the issuer's CN in the end-entity certificate is GTS CA 1C3, the subject's CN in the last intermediate certificate must be GTS CA 1C3, too. The subject's CN of the end-entity certificate must correspond to the requested website. A wildcard is often included. If the client requests www.google.com, the subject's CN can be *.google.com. Also, the current time must be in the certificate's validity period. If the certificate is not yet effective or has expired, the certificate is considered invalid.

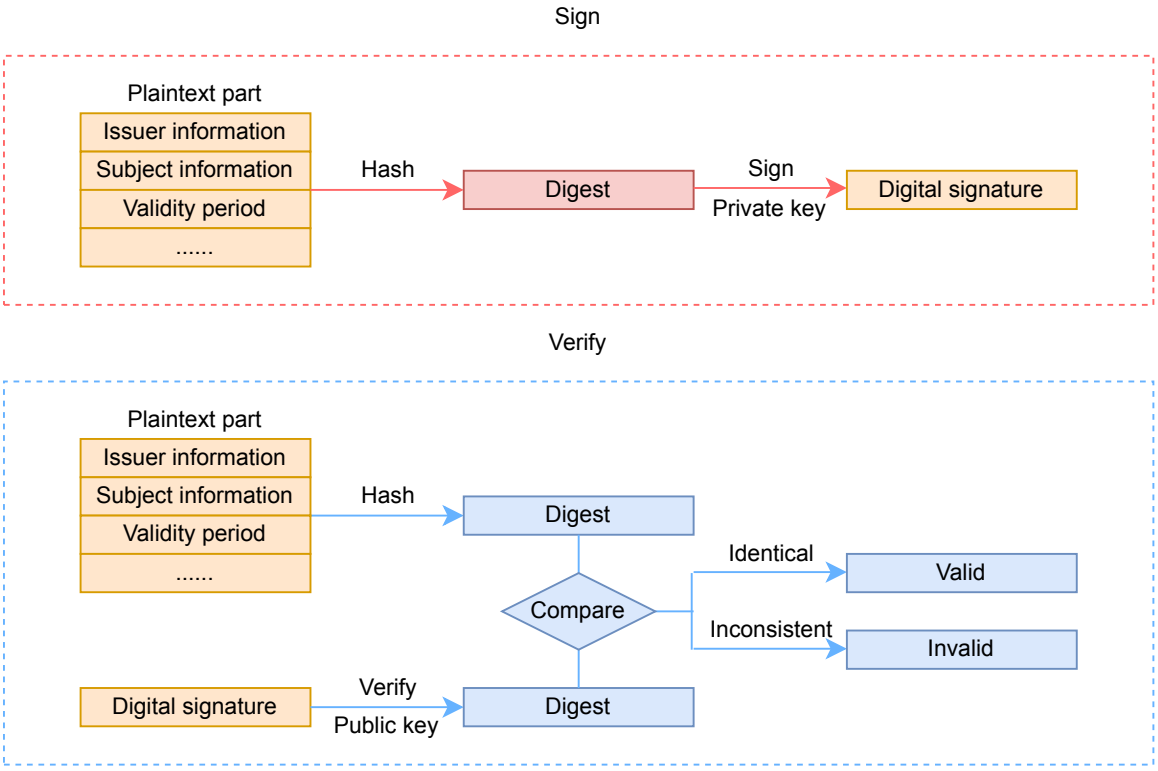

Figure 2.2: The signing and verifying of a certificate

The signing and verifying of the digital certificate are based on a public key algorithm, such as RSA or ECDSA. The steps are shown in Figure 2.2. When signing, the plaintext part of a certificate is hashed. Then, the message digest is signed by the public key algorithm and the CA's private key. The generated digital signature becomes a part of the certificate. When verifying, the client will also hash the plaintext part and get a digest. The digital signature is verified with the public key included in the upper-level certificate. This will recover the original digest. The two digests are compared. If they are identical, the signature is considered valid. The owner of a certificate holds its private key. Only the owner can sign with its private key, but everyone can verify the signature with the public key. The private key must be

well-kept. If the private key is leaked, the attacker can fake the owner's identity.

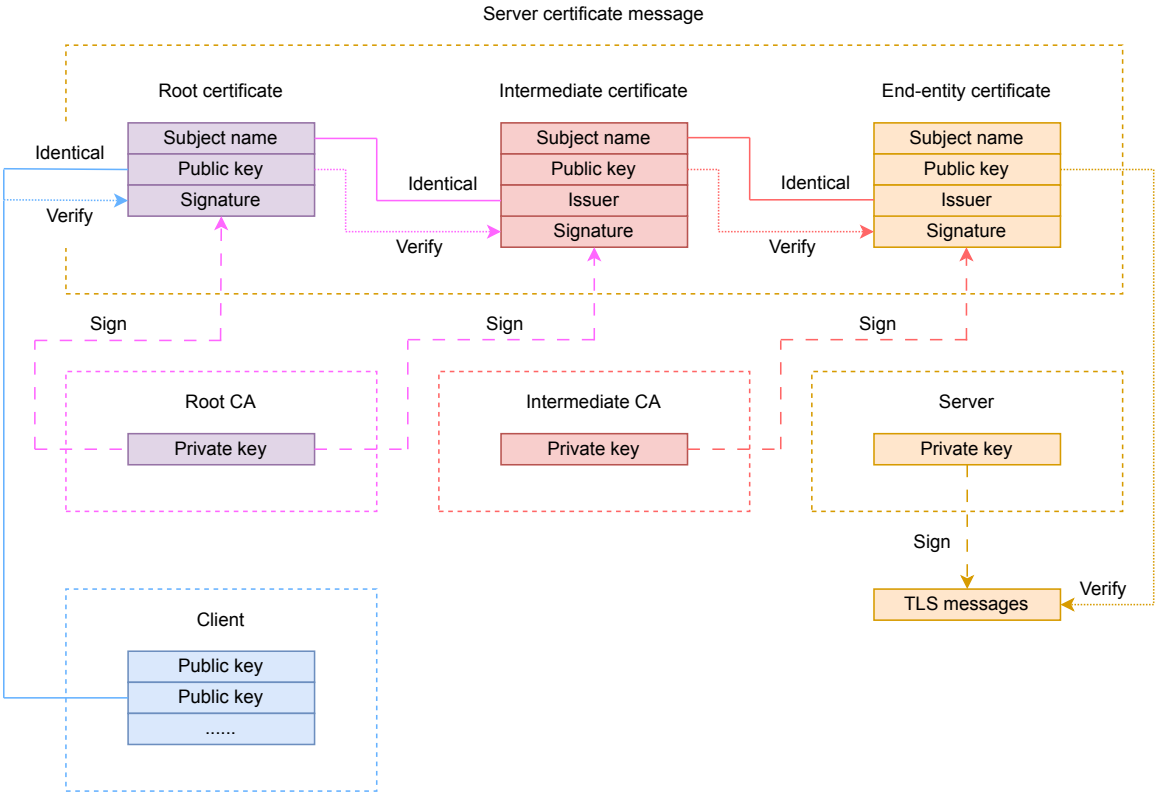

Figure 2.3: The structure of a certificate chain

The digital certificate sent by the server is usually a certificate chain. Figure 2.3 shows the typical structure of a certificate chain. The certificate chain includes multiple certificates, from the end-entity certificate, one or more intermediate certificates, and a root certificate. The end-entity certificate belongs to the server, while other certificates belong to the respective certificate authorities (CA). Each certificate, except the root certificate, is issued and signed by the upper-level CA.

The root certificate is self-signed. It is signed with its private key and can also be verified with its public key. The client (usually a browser) always has the public keys of

some trusted root CAs. The root certificate's public key should be in the client's list, and the root certificate can be verified. The remaining certificates are verified with the public key of their respective upper-level certificates. All certificates in the chain must be valid. If one or more certificates in the chain are invalid, the verification fails. Last, verifying the end-entity certificate's private key is also necessary. The verification is different from other certificates in the chain. In TLS 1.3, the private key signs the past handshake messages and can be verified with the corresponding public key, as shown in Figure 2.3. In TLS 1.2, the certificate participates in the key exchange, and no additional verification is needed.

A server may host multiple websites simultaneously. This is called virtual hosting. Virtual hosting is divided into two kinds: name-based and IP-based. IP-based virtual hosting provides a different IP address for each website. Essentially, this is no different from using multiple servers for the client. Name-based virtual hosting hosts multiple websites on the same IP address and distinguishes each with its hostname. If HTTPS is applied, each website has its certificate, and the server has to pick the correct one. The client must send a Server Name Indication (SNI) containing the hostname [41]. Then, the server can pick the corresponding certificate. For the servers hosting only one website, SNI is theoretically optional. However, the client can never judge if the server only hosts the requested website. SNI will be introduced in detail in the next chapter.

Chapter 3

TLS handshake details

In this chapter, the steps of TLS handshake will be introduced in detail, as our proposed protocol is based on it. A brief introduction has been given in Chapter 2. Compared to TLS 1.2, many changes have been made to the handshake protocol in TLS 1.3. As a result, TLS 1.2 and 1.3 are divided into two sections.

3.1 TLS 1.2 handshake

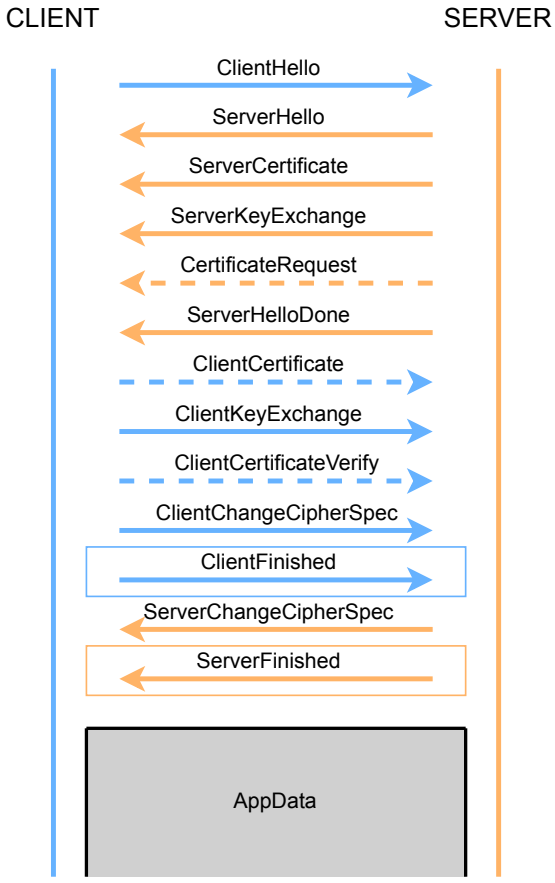

Figure 3.1: TLS 1.2 handshake

Figure 3.1 shows the complete steps of a TLS 1.2 handshake. The messages sent by the client are marked as blue, while those sent by the server are marked as orange. The messages in dotted lines are optional and seldom sent in real applications. Some other steps (like ServerCertificate) are also optional in theory but are almost always sent in practice. The messages in the boxes are encrypted. Next, each step will be explained in detail.

3.1.1 ClientHello

Record header			Message body		
Content type	Version	Length	Handshake type	Message length	Version
0x16	0x0303	2 bytes	0x01	3 bytes	0x0303

Message body					
Random	Session ID length	Session ID	Cipher suite length	Cipher suites	Compression length
32 bytes	1 byte	SID len bytes	2 bytes	CS len bytes	1 byte

Message body					
Compression	Extensions length	Extension type	Extension length	Extension body	.....
Comp. len bytes	2 bytes	2 bytes	2 bytes	Ext length bytes	.....

Figure 3.2: The message format of TLS 1.2 ClientHello

As a sign of the beginning of a handshake, the client always sends a ClientHello message. The message structure is shown in Figure 3.2. All TLS packets have a 5 bytes record header. The first byte indicates the content type, which is 0x16 for TLS 1.2 handshake packets. Version 0x0303 means TLS 1.2. The length is a 2 bytes unsigned integer, which represents the length in bytes of the remaining message after the record header.

Strictly speaking, the server may send another message called HelloRequest before ClientHello. However, it only appears when the server requests a renegotiation, not in an initial handshake. A renegotiation can be performed after a TLS connection has been successfully established, but some extra operation is required for either side. For example, a peer wants to change the cipher suite, or the server wants to request a peer certificate that has yet to be obtained. Renegotiation has been proven to be a vulnerability and prone to man-in-the-middle attacks [42] and denial of service (DoS) attacks [43]. As a result, most major implementations have disabled this feature.

TLS 1.3 has wholly given up renegotiation [34].

The message body of ClientHello includes the highest TLS version the client supports, a random number, a session ID, and a list of available cipher suites and extensions. The handshake type is 0x01, which means ClientHello. The length is a 3 bytes unsigned integer. It is also the length of the remaining part, just like the length in the record header. The version must be TLS 1.2, or in hexadecimal, 0x0303, in a successful TLS 1.2 handshake. Even if the client supports TLS 1.3, the version here is still set as 1.2, and additional extensions are added to show the difference. If the server only supports TLS 1.2, it will reply with a ServerHello message of TLS 1.2. Then, both sides perform a TLS 1.2 handshake. The session ID is used for session resumption. A list of supported cipher suites is included for the server to choose from. They need at least one common cipher suite to continue the handshake. Although a few compression methods are defined [44], compression is often disabled by default due to specific attacks against TLS-level compression [45]. In this case, the compression method length is 1, and the only compression method is NULL (0x00).

Other optional information is placed in the extensions. The extension length indicates the length of all extensions. Each extension has three parts: extension type, length, and body. Server Name Indication (SNI) is the focus here. When TLS is used for HTTPS, an SNI is always included in the extensions. Because of its use in censorship, its details are discussed in Section 3.3.

Record header			Message body		
Content type	Version	Length	Handshake type	Message length	Version
0x16	0x0303	2 bytes	0x02	3 bytes	0x0303

Message body					
Random	Session ID length	Session ID	Cipher suite	Compression length	Compression
32 bytes	1 byte	SID len bytes	2 byte	1 byte	Comp. len bytes

Message body				
Extensions length	Extension type	Extension length	Extension body	.....
2 bytes	2 bytes	2 bytes	Ext length bytes	.....

Figure 3.3: The message format of ServerHello

3.1.2 ServerHello

In response, the server sends a ServerHello message to the client. Its structure is the same as that of ClientHello. The handshake type is 0x02, which means ServerHello. The server generates its own random number. The session ID is copied from ClientHello. The cipher suite is chosen from the ClientHello list and will be used for this connection. If the server does not support any of the client's cipher suites, it will send an error message instead. The extensions in ServerHello do not include SNI. The rest are similar to ClientHello's and will not be repeated here.

3.1.3 ServerCertificate

In most cases, the server sends a ServerCertificate message right after the ServerHello. This message is optional in the protocol, but it is essential for the client to identify the server in real applications. The message format is shown in Figure 3.4. This message contains all the certificates in the server's certificate chain, from the end-entity certificate to the root certificate. The client can verify the certificate and may abort the connection if it is invalid. The certificate's public key also participates in

Record header			Message body		
Content type	Version	Length	Handshake type	Message length	Certificates length
0x16	0x0303	2 bytes	0x0B	3 bytes	3 bytes

Message body				
Certificate length	End-entity cert.	Certificate length	Intermediate cert.	.....
3 bytes	Cert. length bytes	3 bytes	Cert. length bytes	.....

Message body	
Certificate length	Root cert.
3 bytes	Cert. length bytes

Figure 3.4: The message format of the Certificate message

the key exchange. The ServerCertificate is sent before the key exchange is completed and is not encrypted. In an HTTPS session, the domain name requested by the client is usually the subject's common name of the end-entity certificate. Occasionally, it is placed in an extension, the certificate subject alternative name instead. In either case, as long as the attacker intercepts this message, they will learn the website the client is requesting.

3.1.4 ServerKeyExchange

Record header			Message body		
Content type	Version	Length	Handshake type	Message length	Message body
0x16	0x0303	2 bytes	0x0C	3 bytes	Msg. len. bytes

Figure 3.5: The message format of ServerKeyExchange

The server sends this message when the key exchange method of the chosen cipher suite is DHE_DSS, DHE_RSA, DHE_ECC, or DH_anon [31]. Generally, this message contains additional information used by the Diffie-Hellman protocol, and for

DHE_RSA, a digital signature is required. If the elliptic curve cryptography (ECC) is applied, the curve information is also sent. The format is shown in Figure 3.5. The format of the message body varies significantly from case to case and is not shown in the figure. Since the public key of DHE is randomly generated, this message is not related to the server's identity.

3.1.5 CertificateRequest

Record header			Message body	
Content type	Version	Length	Handshake type	Message length
0x16	0x0303	2 bytes	0x0D	3 bytes

Message body					
Cert. type len	Cert. type	Sig. alg. len	Sig. algorithm	Cert. auth. len	Cert. auth.
1 byte	Cert. type len bytes	2 bytes	Sig. alg. len bytes	2 bytes	Cert. auth. len bytes

Figure 3.6: The message format of CertificateRequest

This message is only sent when the server needs to identify the client with a certificate. The message format is shown in Figure 3.6. Considering that almost no website has adopted this authentication method [46], it rarely appears. Compared to a client certificate, an authentication in application layer is usually considered more convenient.

3.1.6 ServerHelloDone

Record header			Message body	
Content type	Version	Length	Handshake type	Message length
0x16	0x0303	0x0004	0x0E	0x000000

Figure 3.7: The message format of ServerHelloDone

The server will send a ServerHelloDone message after the few messages above have been sent. As the name suggests, the messages sent by the server have ended, and the server will wait for the client to respond. It only includes the message type and a length of 0, as shown in Figure 3.7.

3.1.7 ClientCertificate

If the server requests the client to provide its certificate, the client must send a ClientCertificate after ServerHelloDone. The message format is identical to the ServerCertificate in Section 3.1.3. The certificate type and authority must be chosen from the list in CertificateRequest. This message is also unencrypted: the attacker can learn the identity of the user from this message.

3.1.8 ClientKeyExchange

A ClientKeyExchange is always sent, regardless of the cipher suite. If the RSA key exchange algorithm is used, it will contain the RSA-encrypted premaster secret. If the Diffie-Hellman algorithm is used, it sends the DH public key or an empty message, depending on whether the DH algorithm is ephemeral or static. The difference be-

tween the two is whether to use the same Diffie-Hellman private keys in each session. Its message format is the same as ServerKeyExchange, though the message type is 0x10. This message is generated by the client and not related to the server's identity.

3.1.9 CertificateVerify

Record header			Message body	
Content type	Version	Length	Handshake type	Message length
0x16	0x0303	2 bytes	0x0F	3 bytes

Message body			
Hash algorithm	Signature algorithm	Signature length	Signature
1 byte	1 byte	2 bytes	Signature length bytes

Figure 3.8: The message format of CertificateVerify

If a client certificate is requested and sent, and the client certificate has signing capability, this message will follow the ClientKeyExchange. Its format is shown in Figure 3.8. This message only includes a digital signature. All past handshake messages are hashed together and signed with the certificate private key. The hash and signature algorithm is chosen from the lists in the CertificateRequest message. When the server receives this message, it will verify the digital signature with the certificate's public key and send a fatal error message to abort the connection if the signature is invalid.

3.1.10 ClientChangeCipherSpec

After the key exchange has been completed, the client sends ClientChangeCipherSpec. Its format is shown in Figure 3.9. All the following messages from the client side will be encrypted. The content type in the record header is ChangeCipherSpec (0x14),

Record header			Message body
Content type	Version	Length	0x01
0x14	0x0303	0x0001	

Figure 3.9: The message format of ChangeCipherSpec

not handshake. The message body is fixed as a 1.

3.1.11 ClientFinished

Record header			Encrypted message body		
Content type	Version	Length	Handshake type	Message length	Verify_data
0x16	0x0303	2 bytes	0x14	0x00000C	12 bytes

Figure 3.10: The message format of ClientFinished

This is the last handshake message sent by the client, and it is the first message encrypted with the negotiated algorithms and keys. All encrypted messages also have a MAC or AEAD tag, which is not shown in Figure 3.10. It includes 12 bytes of **verify_data**, which is generated by the master secret, a string which is "client finished" for ClientFinished or "server finished" for ServerFinished, and the hash value of all handshake messages. If the expected value calculated by the server differs from the received value in the ClientFinished, the server will send an alert message and abort the connection.

3.1.12 ServerChangeCipherSpec

The format and effect of this message are the same as ClientChangeCipherSpec, though it is sent by the server.

3.1.13 ServerFinished

ServerFinished has the same effect as ClientFinished. After the client receives the ServerFinished and parses it, the handshake is over, and the client will start to send application data. All of the application data messages sent from both sides are now encrypted.

3.2 TLS 1.3 handshake

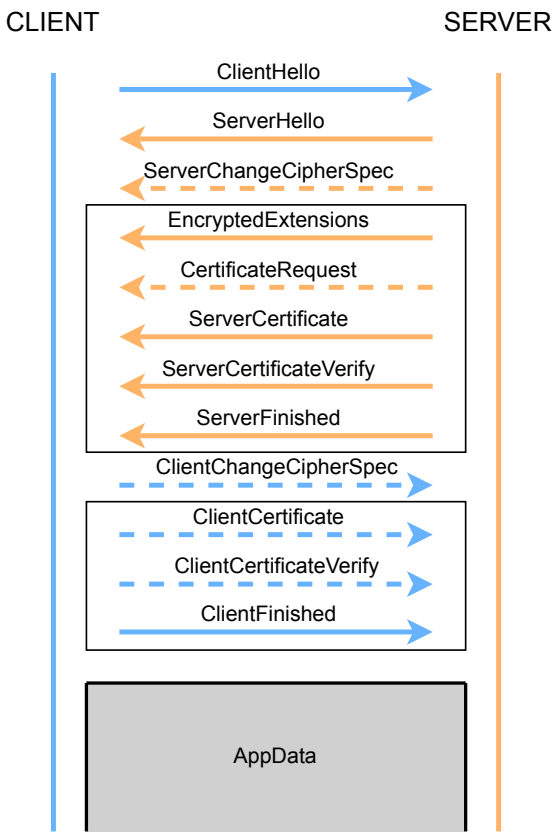

Figure 3.11: TLS 1.3 handshake

Figure 3.11 shows the complete steps of a TLS 1.3 handshake. Compared to TLS 1.2, a few changes have been made. The key exchange can be performed with only ClientHello and ServerHello, so no additional key exchange message is required. The CertificateVerify messages become compulsory if the corresponding certificate is sent first. Another difference is that once the key exchange has been done on either side, all messages from that side will be marked as application data in the record header. This makes it more challenging to match the intercepted data packets with the handshake steps. Moreover, only one roundtrip is needed in a TLS 1.3 handshake, while in TLS 1.2, the number is two. This slightly improves performance. Next, the specific differences in each step will be explained below.

3.2.1 ClientHello

As mentioned in Section 3.1.1, the ClientHello of TLS 1.3 differs from the ClientHello of TLS 1.2 only in extensions. The record header and message body versions are still TLS 1.2 for backwards compatibility. A new extension, which is called the supported versions, is compulsory in TLS 1.3. It includes the TLS versions supported by the client, and they are usually TLS 1.3 and 1.2 since versions before TLS 1.1 have been deprecated. If the server supports TLS 1.3, it will proceed with TLS 1.3; if not, it will switch to TLS 1.2 and send the handshake messages according to the standard of TLS 1.2. Another necessary extension is **key share**. It undertakes the function of ClientKeyExchange.

3.2.2 ServerHello

Like ClientHello, the ServerHello is the same as in TLS 1.2 except for the extensions. Again, the supported versions and **key share** extensions are included. This time, only one version, TLS 1.3, should appear in the supported versions as a TLS 1.3 handshake. The ServerKeyExchange message in TLS 1.2 is replaced by a new extension, the **key share**. When this message is sent, the key exchange on the server side has been completed.

3.2.3 ServerChangeCipherSpec

The purpose and format of this message are identical to the TLS 1.2's, but it is optional in TLS 1.3. It is only sent when the server and client work in middlebox compatibility mode [34], which improves the compatibility with TLS 1.2 middleboxes.

3.2.4 EncryptedExtensions

Record header			Encrypted message body		
Content type	Version	Length	Handshake type	Message length	Extensions length
0x18	0x0303	2 bytes	0x08	3 bytes	2 bytes

Encrypted message body			
Extension type	Extension length	Extension body	.....
2 bytes	2 bytes	Ext length bytes	.....

Figure 3.12: The message format of EncryptedExtensions

EncryptedExtensions is a new message defined in TLS 1.3, and it must be sent, though the list of extensions can be empty. This is the first encrypted message from the server, and it only includes the extensions that need to be protected, such as Heartbeat. The

Heartbeat Extension is used to keep the connection alive by sending periodic keep-alive messages. The extensions in EncryptedExtensions are not included in the key exchange or associated with certificates [34].

3.2.5 CertificateRequest

Like TLS 1.2, CertificateRequest is sent only when the server needs to request a certificate from the client. A few changes have been made to the format of this message, but they are not relevant to this study. Also, it is encrypted in TLS 1.3.

3.2.6 ServerCertificate

The format of this message has not changed in TLS 1.3, but it is encrypted and does not participate in the key exchange. The attacker cannot get the certificate information by intercepting the data packet unless the encryption is broken. However, it is still prone to active probing: the attacker can try to access the website and get the certificate.

3.2.7 ServerCertificateVerify

This message has the same function as the CertificateVerify in TLS 1.2. However, it is also compulsory for the server certificate in TLS 1.3 since the server certificate does not participate in the key exchange. The format does not change, but the message is encrypted since it must be sent after ServerCertificate.

3.2.8 ServerFinished and ClientFinished

These two messages are still used to demonstrate that the handshake has finished on the corresponding side. The generation of the **verify_data** has changed: the size is not fixed as 12 bytes, but is the full output of the HMAC.

3.3 SNI-based censorship

The format of Server Name Indication (SNI) is shown in Figure 3.13. The server name type can only be 0x00 in practice, which means hostname. The other value, 0xFF, is reserved [47]. The server name part is the hostname that the client is currently requesting. Because the ClientHello message is not encrypted, the SNI can be intercepted easily.

SNI extension body			
SN list length	Server name type	Server name length	Server name
2 bytes	1 byte	2 bytes	SN length bytes

Figure 3.13: The message format of the SNI extension

A censor can identify the server by parsing the ClientHello, so SNI-based censorship is deployed in most countries where internet censorship exists [17, 18, 19]. Taking the GFW as an example, all ClientHello messages are filtered, and like HTTP keyword filtering, the censorship middlebox implements TCP packet injection if the SNI is on the blacklist. It sends TCP RST packets to both sides to terminate the connection. One research direction is to confront TCP packet injection [48, 49, 50, 51], but

their rivals are also evolving. Two SNI-based censorship middleboxes are deployed by the GFW [17]. Even if the first one is not working, the second one may still be effective. More seriously, the GFW is piloting an SNI whitelist in several cities [52]. All websites except those on the whitelist are blocked. Residual censorship is applied in China, Iran, and Kazakhstan [53]: once a blacklisted SNI is detected, all connections between the target client's and server's IP address and port will be reset. This lasts for 60 to 300 seconds. In order to bypass SNI-based censorship, encryption or spoofing of SNI is required. Many solutions have been proposed for this, and they will be introduced in detail and compared with SNIT in Chapter 8.

Chapter 4

SNIT: SNI Tunneling

In this research, we propose a modification, named SNIT (SNI Tunneling), to TLS 1.2 and TLS 1.3. With SNIT, a secret website can falsify its SNI and certificate as an innocuous website. From the censor's point of view, accessing a secret website with SNIT appears the same as the original handshake protocol. Compatibility for unmodified clients and servers is maintained. For example, a server can host *public.com* and *secret.com* simultaneously, but *secret.com* is accessible only via SNIT. The client can validate the certificate of *secret.com* in the handshake. Next, a detailed description of the handshake steps will be given, using *public.com* as the cover domain and *secret.com* as the covert domain.

4.1 SNIT on TLS 1.2

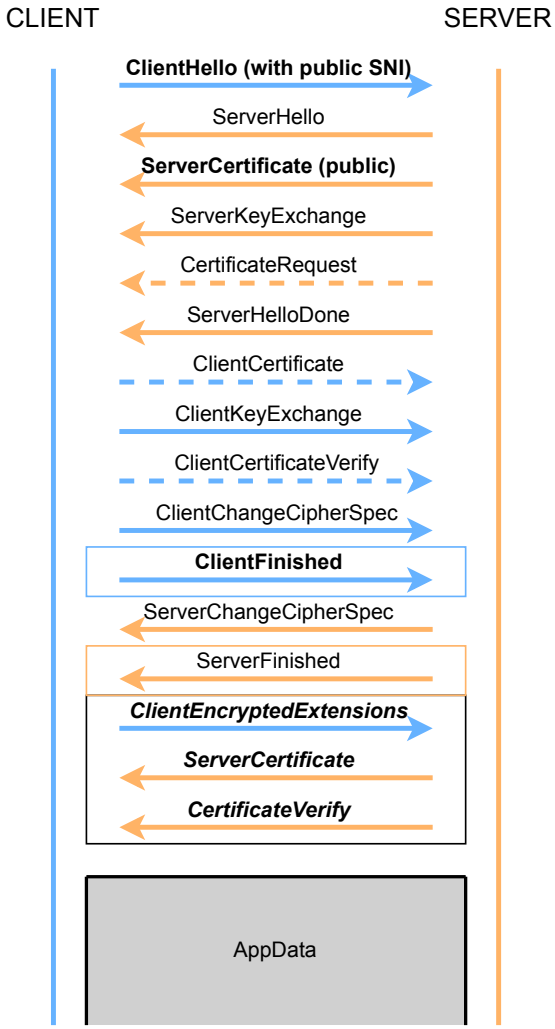

Figure 4.1: SNIT on TLS 1.2

The steps of SNIT on TLS 1.2 are shown in Figure 4.1. The modified steps are marked in bold, while the additional steps are marked in bold and italics. The black box indicates that the messages inside are encrypted and marked as application data. The rest of the annotations are the same as in Figure 3.1. The unencrypted SNI and

certificate are replaced with an innocuous website, and the real SNI and certificate are sent after the end of the normal handshake process. The steps after ServerFinished are all marked as application data in the record header since no encrypted handshake messages should appear here. The part before this looks the same as the original protocol, but some modifications are made to maintain the compatibility. Because of the additional steps, the handshake needs one more roundtrip. The new and modified handshake steps will be introduced below.

4.1.1 ClientHello

The structure of this message does not change, but the SNI is set as a cover website, e.g. *public.com*, whether the client is accessing *public.com* with the original protocol or *secret.com* with SNIT. One or more corresponding public websites protect each secret website. Since the SNI is always the cover domain, SNI-based censorship does not work anymore. Like Domain Fronting and BlindTLS (see Chapter 8), the public server name can only be acquired out-of-band.

4.1.2 ServerCertificate

The first ServerCertificate, right after ServerHello, belongs to *public.com*. Like ClientHello, this message is the same in both cases. It must be a valid certificate of *public.com*. Although *public.com* is not the website the client wants to access, the client should still validate the certificate, as the server might be forged by the censor. This certificate participates in the key exchange.

4.1.3 ClientCertificate

This message is no different from the original protocol, but the client certificate should only be used to authenticate to *public.com*. The client can be identified by *secret.com* by other methods, which will be explained in the next step.

4.1.4 ClientFinished

Record header			Encrypted message body		
Content type	Version	Length	Handshake type	Message length	Verify_data XOR pw
0x16	0x0303	2 bytes	0x15	0x00000C	12 bytes

Figure 4.2: The message format of ClientFinished of SNIT

Record header			Encrypted message body		
Content type	Version	Length	Handshake type	Message length	Verify_data
0x16	0x0303	2 bytes	0x14	0x00000C	12 bytes

Figure 4.3: The message format of ClientFinished of the original protocol

This message is used to distinguish the two protocols, as shown in Figure 4.4. If the client requests *public.com* with the original protocol, this message does not change, and both sides will continue with the original protocol. If the client uses SNIT, the message format is shown in Figure 4.2. The message format of the original ClientFinished is also shown in Figure 4.3 for comparison. The handshake type is set as 0x15, which is named **fake_finished**, not 0x14 or **finished** in the original finished messages. The generating algorithm of the **verify_data** does not change. It is still 12 bytes, but XORed with a 96b password. The password is set by the server, and the

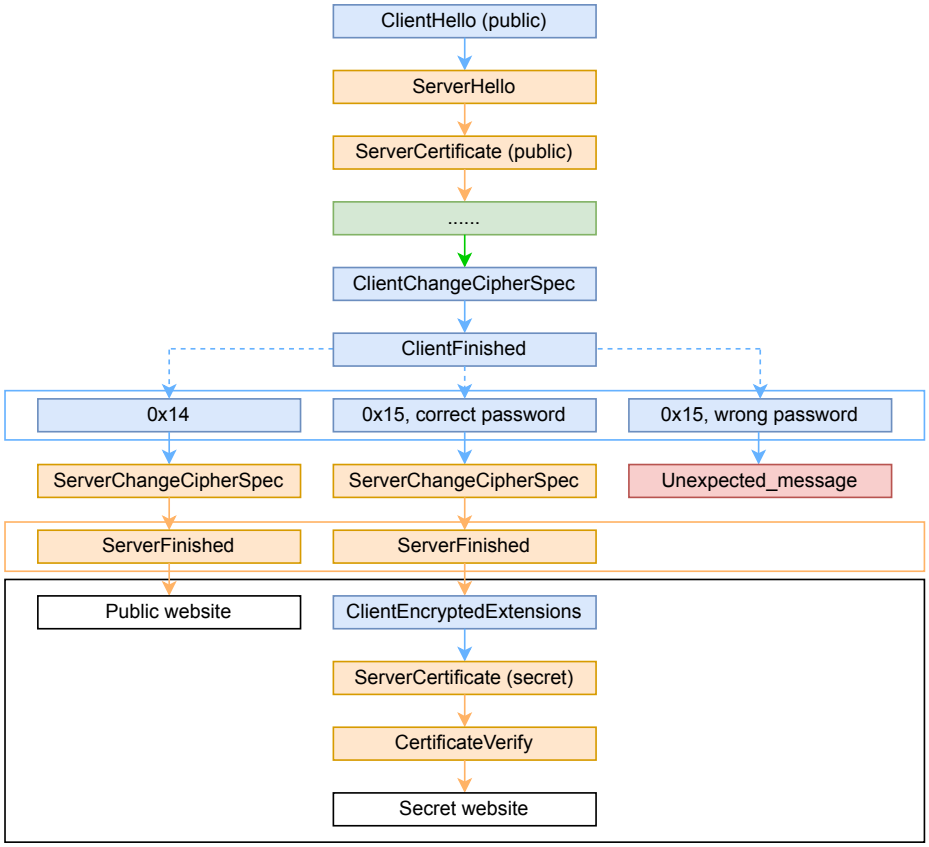

Figure 4.4: The flowchart of SNIT

client must learn the correct password out-of-band. When verifying this message, the server will send an **unexpected_message** error to the client if the **verify_data** is not correct, whether the password is wrong or the **verify_data** itself is wrong. This is done to prevent active probing. The ServerFinished message does not change.

4.1.5 ClientEncryptedExtensions

The format of this message is identical to the EncryptedExtensions message of TLS 1.3, though it has different uses. This message is encrypted and marked as application data in the record header, like all following handshake messages. Currently, only one

extension, the server name indication, is supported. Its use is the same as the SNI in the original protocol to let the server send the corresponding certificate. The censor cannot get *secret.com* from the handshake unless the encryption is broken.

4.1.6 ServerCertificate

This second certificate belongs to *secret.com*. The message format is also the same as the first one, except it is encrypted. Then, the client can validate the certificate and the server's identity. The key exchange will not be performed again since an additional key exchange does not improve security.

4.1.7 CertificateVerify

Since the secret certificate does not participate in the key exchange, an additional verification of the certificate's private key is needed. The use and format of this message are identical to the CertificateVerify of the client's certificate. A digital signature is generated by the certificate's private key and all past handshake messages.

4.2 SNIT on TLS 1.3

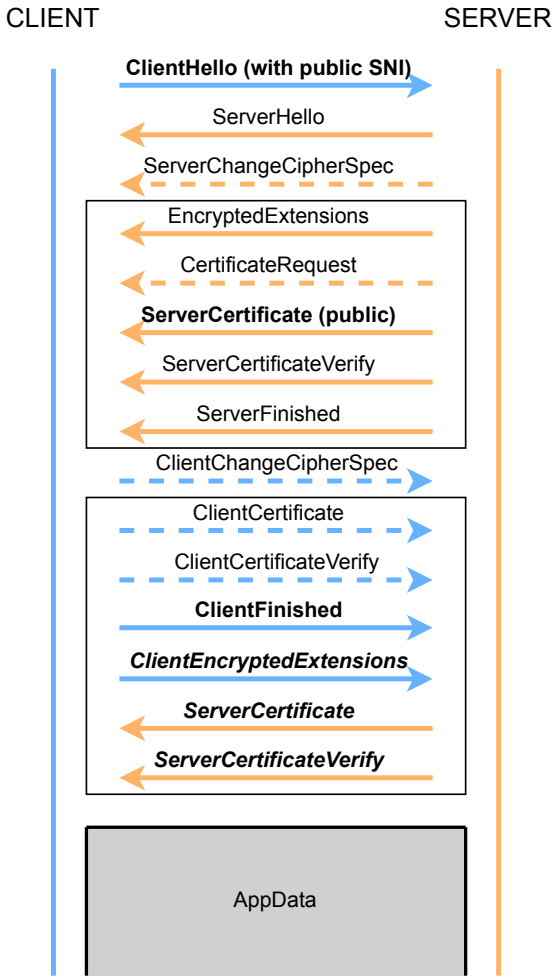

Figure 4.5: SNIT on TLS 1.3

The steps of SNIT on TLS 1.3 are shown in Figure 4.5. The annotations are the same as in Figure 4.1. The modifications are similar to those of TLS 1.2. The cover domain is still the SNI in ClientHello and the common name of the first ServerCertificate. Like the original protocol, the certificate is encrypted and does not participate in the key exchange, but the client still needs to validate it. The password in ClientFinished

is still 96b. If the **verify_data** is longer than 96 bits, the password will be XORed with the first 96 bits of the **verify_data**. The use and format of ClientEncryptedExtensions, ServerCertificate, and ServerCertificateVerify do not change.

4.3 Deployment

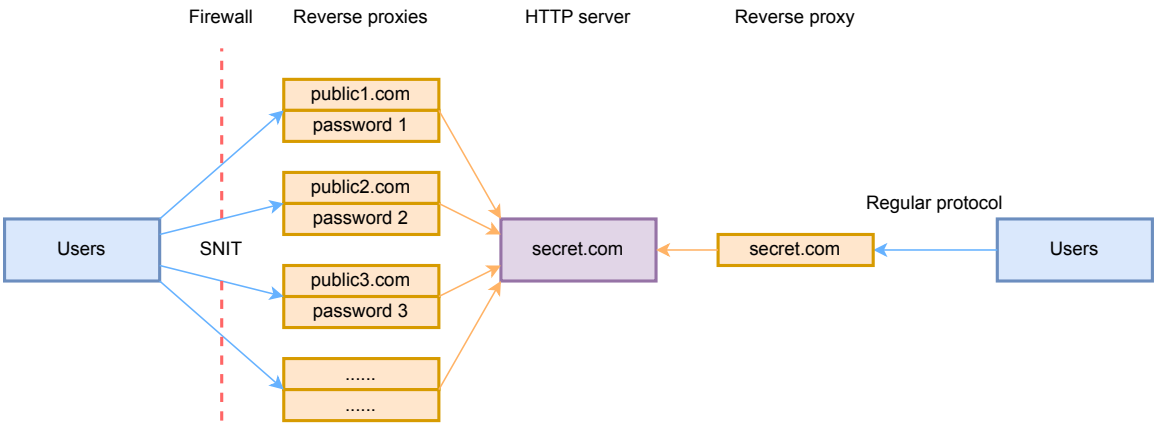

Figure 4.6: Deployment of SNIT using reverse proxies

SNIT is compatible with the original protocol. It can be attached to any TLS server. One feasible deployment mode is shown in Figure 4.6. Here, the origin HTTP server of the secret website is placed outside of the firewall, with reverse proxies supporting both the original TLS handshake and SNIT. Each proxy owns different IP addresses, public SNIs, and passwords. These public websites must be accessible using the original protocol. Also, all proxies must possess the secret key corresponding to the *secret.com* certificate. The others can survive even if one reverse proxy is exposed and blocked. A user inside the firewall needs a public SNI, as well as the password and the secret SNI, to perform the handshake. The IP addresses of the proxies can

be directly resolved from the public SNIs. After the handshake, the proxy forwards the unencrypted application data to the HTTP server of *secret.com*, and the user has access to *secret.com*. Also, another reverse proxy that only supports the original protocol can be applied. This improves performance and convenience for the users outside the firewall.

SNIT can also be deployed on CDNs. A CDN hosts many domains and owns their corresponding certificates. This allows it to combine any public and secret websites at will. A public website can be used to protect multiple secret websites and vice versa. Many-to-many is also possible. This brings huge flexibility. However, CDN is a double-edged sword. If IP blocking is applied, too many websites on the CDN will be affected. On the one hand, this leads to severe overblocking and goes against the target of screening. The censor may not be willing to block all normal websites on the CDN only because of a small number of sensitive websites. On the other hand, once the censor thinks it is worth using IP blocking, all secret websites hosted on specific IP addresses will be blocked, including those the censor does not know.

4.4 Password distribution

Distribution is another essential part of SNIT. In order to access a secret website, a public SNI and the matched password must be distributed out-of-band. This is the most significant disadvantage of this protocol, though it is common for similar approaches such as Domain Fronting [54] and HTTPT [1] (see Chapter 8). Out-of-band distribution increases the chance of being discovered by the censor. In real applications, different websites have different scales and security levels. Security and

convenience can be weighed differently in different situations.

Here, a feasible method for websites with an average level of security requirements is given. All users are divided into different user groups. The number of groups is precisely the number of public websites. All users in a user group are given one public SNI and the password. The public websites work independently. They can be added, modified, or removed at any time. A user does not know the other pairs, even their number. If an insider discloses one pair to the censor and makes it blocked, only one user group is affected, and the others can survive. Furthermore, the censor can never confirm that all public websites are blocked. As a result, completely blocking a large secret website requires many insiders. For websites with higher demand for security, a bridge distribution system can be applied [55, 56, 57]. In these studies, the bridge distribution system is based on a user reputation system. When a user joins the system, they receive a specific number of bridges. Credits can be earned based on the uptime of these bridges and lost if one of them is blocked. A user with too low reputation is considered as an insider. This prevents them from gaining additional bridges.

In this chapter, our new protocol, SNIT, is introduced. Its difference with the original TLS 1.2 and 1.3 protocol is explained in detail. The deployment and password distribution are also discussed.

Chapter 5

Implementation

5.1 Mbed TLS

Our prototype implementation is based on Mbed TLS. Mbed TLS (previously PolarSSL) is a lightweight TLS library [58]. It is developed by ARM. As a Trusted Firmware project of Arm architecture, Mbed TLS is the preferred implementation of Arm specifications. Its small code and memory footprint make it very suitable for embedded systems. Mbed TLS has been adopted by many development frameworks and operating systems, such as ESP-IDF [59] and RIOT [60].

```
/*
 * SSL state machine
 */
typedef enum
{
    MBEDTLS_SSL_HELLO_REQUEST,
    MBEDTLS_SSL_CLIENT_HELLO,
    MBEDTLS_SSL_SERVER_HELLO,
    MBEDTLS_SSL_SERVER_CERTIFICATE,
    MBEDTLS_SSL_SERVER_KEY_EXCHANGE,
    MBEDTLS_SSL_CERTIFICATE_REQUEST,
    MBEDTLS_SSL_SERVER_HELLO_DONE,
    MBEDTLS_SSL_CLIENT_CERTIFICATE,
    MBEDTLS_SSL_CLIENT_KEY_EXCHANGE,
    MBEDTLS_SSL_CERTIFICATE_VERIFY,
    MBEDTLS_SSL_CLIENT_CHANGE_CIPHER_SPEC,
    MBEDTLS_SSL_CLIENT_FINISHED,
    MBEDTLS_SSL_SERVER_CHANGE_CIPHER_SPEC,
    MBEDTLS_SSL_SERVER_FINISHED,
    MBEDTLS_SSL_FLUSH_BUFFERS,
    MBEDTLS_SSL_HANDSHAKE_WRAPUP,
    MBEDTLS_SSL_HANDSHAKE_OVER,
    MBEDTLS_SSL_SERVER_NEW_SESSION_TICKET,
    MBEDTLS_SSL_SERVER_HELLO_VERIFY_REQUEST_SENT,
    MBEDTLS_SSL_HELLO_RETRY_REQUEST,
    MBEDTLS_SSL_ENCRYPTED_EXTENSIONS,
    MBEDTLS_SSL_CLIENT_CERTIFICATE_VERIFY,
    MBEDTLS_SSL_CLIENT_CCS_AFTER_SERVER_FINISHED,
    MBEDTLS_SSL_CLIENT_CCS_BEFORE_2ND_CLIENT_HELLO,
    MBEDTLS_SSL_SERVER_CCS_AFTER_SERVER_HELLO,
    MBEDTLS_SSL_SERVER_CCS_AFTER_HELLO_RETRY_REQUEST,
    MBEDTLS_SSL_CLIENT_ENCRYPTED_EXTENSIONS,
    MBEDTLS_SSL_SERVER_ENCRYPTED_CERTIFICATE,
    MBEDTLS_SSL_SERVER_ENCRYPTED_CERTIFICATE_VERIFY
}
mbedtls_ssl_states;
```

Figure 5.1: The additional messages added by SNIT, highlighted in green

The table of states is shown in Figure 5.1. The three additional messages introduced in Chapter 4 are highlighted. They are the same in both TLS 1.2 and TLS 1.3, and no other steps are added.

```
#define MBEDTLS_SSL_HS_HELLO_REQUEST            0
#define MBEDTLS_SSL_HS_CLIENT_HELLO             1
#define MBEDTLS_SSL_HS_SERVER_HELLO             2
#define MBEDTLS_SSL_HS_HELLO_VERIFY_REQUEST     3
#define MBEDTLS_SSL_HS_NEW_SESSION_TICKET       4
#define MBEDTLS_SSL_HS_ENCRYPTED_EXTENSIONS     8 // NEW IN TLS 1.3
#define MBEDTLS_SSL_HS_CERTIFICATE             11
#define MBEDTLS_SSL_HS_SERVER_KEY_EXCHANGE     12
#define MBEDTLS_SSL_HS_CERTIFICATE_REQUEST     13
#define MBEDTLS_SSL_HS_SERVER_HELLO_DONE       14
#define MBEDTLS_SSL_HS_CERTIFICATE_VERIFY      15
#define MBEDTLS_SSL_HS_CLIENT_KEY_EXCHANGE     16
#define MBEDTLS_SSL_HS_FINISHED                20
#define MBEDTLS_SSL_HS_FAKE_FINISHED           21
#define MBEDTLS_SSL_HS_MESSAGE_HASH           254
```

Figure 5.2: The new handshake type

```
#if defined(MBEDTLS_SSL_SRV_C)
    mbedtls_ssl_hs_cb_t MBEDTLS_PRIVATE(f_cert_cb);  /*!< certificate selection callback */
#endif /* MBEDTLS_SSL_SRV_C */

#if defined(MBEDTLS_KEY_EXCHANGE_CERT_REQ_ALLOWED_ENABLED)
    const mbedtls_x509_crt *MBEDTLS_PRIVATE(dn_hints);/*!< acceptable client cert issuers    */
#endif
    uint128_t pw;
};
```

Figure 5.3: The password

Figure 5.2 shows the new handshake type, fake_finished. Its purpose is to distinguish SNIT, as explained in detail in Section 4.1.4. The password is 96 bits. In our implementation, it is stored in a 128b integer, as shown in Figure 5.3.

```
    else{
        if( ssl->conf->endpoint == MBEDTLS_SSL_IS_CLIENT &&
            ssl->fake_protocol == MBEDTLS_SSL_FAKE_PROTOCOL )
            ssl->state = MBEDTLS_SSL_CLIENT_ENCRYPTED_EXTENSIONS;
        else
            ssl->state++;
    }
```

Figure 5.4: The state switching of TLS 1.2 ClientFinished and ServerFinished, difference highlighted in blue

The format of ClientFinished and ServerFinished are the same. Their parsing or writing is implemented in one function in Mbed TLS. Figure 5.4 shows the state switching of TLS 1.2 ClientFinished and ServerFinished parsing functions as described in Section 4.1.4. The server only parses ClientFinished, and the client only parses ServerFinished. On the server side, the ClientFinished will always switch to the next state, ServerChangeCipherSpec. The state table can be found in Figure 5.1. If the original protocol is used, ServerFinished will switch to `MBEDTLS_SSL_FLUSH_BUFFERS`, the next step in the state table. This means the handshake is finished. Only when the client uses SNIT will it switch to ClientEncryptedExtensions. The state switching of the writing function is no different from the parsing function and will not be repeated.

```
    if( ssl->state == MBEDTLS_SSL_SERVER_CERTIFICATE )
        mbedtls_ssl_handshake_set_state( ssl, MBEDTLS_SSL_CERTIFICATE_VERIFY );
    else
        mbedtls_ssl_handshake_set_state( ssl, MBEDTLS_SSL_SERVER_ENCRYPTED_CERTIFICATE_VERIFY
```

Figure 5.5: The state switching of ServerCertificate

```
if( ssl->state == MBEDTLS_SSL_CERTIFICATE_VERIFY )
    mbedtls_ssl_handshake_set_state( ssl, MBEDTLS_SSL_SERVER_FINISHED );
else
    mbedtls_ssl_handshake_set_state( ssl, MBEDTLS_SSL_FLUSH_BUFFERS );
```

Figure 5.6: The state switching of ServerCertificateVerify

Like ClientFinished, ServerCertificate and ServerCertificateVerify have different state switching under different circumstances. In TLS 1.3, the only ServerCertificate in the original protocol and the public ServerCeritificate in SNIT is defined as `MBEDTLS_SSL_SERVER_CERTIFICATE`. The next state is always `MBEDTLS_SSL_CERTIFICATE_VERIFY`. If it is the secret ServerCertificate, the next step is defined as `MBEDTLS_SSL_SERVER_ENCRYPTED_CERTIFICATE_VERIFY`, which is shown in Figure 5.1. In these two cases, the next state of ServerCertificateVerify is ServerFinished or flush buffers, respectively.

```
if( ssl->state == MBEDTLS_SSL_CLIENT_FINISHED )
{
    if( ssl->in_msg[0] == MBEDTLS_SSL_HS_FAKE_FINISHED )
    {
        ssl->fake_protocol = MBEDTLS_SSL_FAKE_PROTOCOL;
        mbedtls_printf( "Client is using the modified TLS 1.3 protocol.\n" );
        *(uint128_t *) expected_verify_data ^= ssl->pw;
    }
    else
    {
        mbedtls_printf( "Client is using the original TLS 1.3 protocol.\n" );
    }
}
```

Figure 5.7: The switching of protocols

If the server supports SNIT, and the message type in ClientFinished is fake_finished, the server will change to SNIT and XOR to **verify_data** with the password.

```
/* Semantic validation */
if( mbedtls_ct_memcmp( buf,
                       expected_verify_data,
                       expected_verify_data_len ) != 0 )
{
    MBEDTLS_SSL_DEBUG_MSG( 1, ( "bad finished message" ) );
    if( ssl->in_msg[0] == MBEDTLS_SSL_HS_FAKE_FINISHED )
    {
        MBEDTLS_SSL_PEND_FATAL_ALERT( MBEDTLS_SSL_ALERT_MSG_UNEXPECTED_MESSAGE,
                                      MBEDTLS_ERR_SSL_UNEXPECTED_MESSAGE );
    }
    else
    {
        MBEDTLS_SSL_PEND_FATAL_ALERT( MBEDTLS_SSL_ALERT_MSG_DECRYPT_ERROR,
                                      MBEDTLS_ERR_SSL_HANDSHAKE_FAILURE );
    }
```

Figure 5.8: The verification of the password

Since the password is XORed to the **verify_data**, they can only be verified together. If the client uses the original protocol and gets the wrong **verify_data**, the server will reply with a decrypt_error. If it uses SNIT and either the **verify_data** or the password is wrong, the server will reply with an unexpected_message instead.

```
    /* Now write the potentially updated record content type. */
    ssl->out_hdr[0] = (unsigned char) ssl->out_msgtype;
    if( ssl->state >= MBEDTLS_SSL_CLIENT_ENCRYPTED_EXTENSIONS          &&
        ssl->state <= MBEDTLS_SSL_SERVER_ENCRYPTED_CERTIFICATE_VERIFY  &&
        ssl->out_msgtype == MBEDTLS_SSL_MSG_HANDSHAKE                  &&
        ssl->transform_in->tls_version == MBEDTLS_SSL_VERSION_TLS1_2   )
        ssl->out_hdr[0] = (unsigned char) MBEDTLS_SSL_MSG_APPLICATION_DATA;
    rec->type = buf[ rec_hdr_type_offset ];

    /* Change fake app data to handshake */

    if( ssl->state >= MBEDTLS_SSL_CLIENT_ENCRYPTED_EXTENSIONS          &&
        ssl->state <= MBEDTLS_SSL_SERVER_ENCRYPTED_CERTIFICATE_VERIFY  &&
        rec->type == MBEDTLS_SSL_MSG_APPLICATION_DATA                  &&
        ssl->transform_in->tls_version == MBEDTLS_SSL_VERSION_TLS1_2   )
        rec->type = MBEDTLS_SSL_MSG_HANDSHAKE;
```

Figure 5.9: The disguised application data in TLS 1.2

In TLS 1.2, the three additional steps are marked as application data, but they

are handshake messages indeed. Their record type must be changed to application data when writing or changed to handshake when parsing. This is necessary to pass the sanity check.

5.2 Hiawatha webserver

In order to evaluate the performance of SNIT in a realistic environment, we also implemented SNIT on a web server, Hiawatha, based on Mbed TLS. We have implemented the core functionality of SNIT, such as the handshake procedure and switching of certificates and websites. The public and secret websites are deployed one-to-one. The reverse proxy in Section 4.3 has not been implemented yet.

Table 5.1 shows the behaviours of the server in different situations. We mimic regular server behaviour unless all SNIT checks pass. Each column indicates a different input or output. The inputs by the client are the public SNI, the secret SNI, and the hostname in the HTTP request. The outputs are the public, secret certificates, and web pages in the HTTP response. The server hosts five domains: *public1.com*, *secret1.com*, *public2.com*, *secret2.com* and *default.com*. The corresponding certificates are self-signed and generated by OpenSSL. There are two groups of public and secret websites. *Secret1.com* and *secret2.com* are protected by *public1.com* and *public2.com*, respectively. Since the reverse proxy is not yet supported, the secret websites cannot be accessed using the original protocol from this server. A server may host more pairs of websites or public websites, but two pairs are enough for the test. *Default.com* applies when an invalid SNI or hostname is received in the HTTP request.

If the client uses the original protocol, it will get the certificate of *public1.com*

Table 5.1: The behaviours of the server in different situations

Protocol	Public SNI	Secret SNI	HTTP req.	Public cert.	Secret cert.	Webpage
Original	Public1.com	-	Public1.com	Public1.com	-	Public1.com
			Other			Default.com
	Secret1.com	-	Any	Default.com	-	Default.com
	Public2.com	-	Public2.com	Public2.com	-	Public2.com
			Other			Default.com
	Secret2.com	-	Any	Default.com	-	Default.com
	Other	-	Any	Default.com	-	Default.com
SNIT	Public1.com	Public1.com	Public1.com	Public1.com	Public1.com	Public1.com
			Other			Default.com
		Secret1.com	Secret1.com		Secret1.com	Secret1.com
			Public1.com			Public1.com
			Other			Default.com
		Other	Public1.com		Public1.com	Public1.com
			Other			Default.com
	Secret1.com	-	-	Default.com	Unexpected message	
	Public2.com	Public2.com	Public2.com	Public2.com	Public2.com	Public2.com
			Other			Default.com
		Secret2.com	Secret2.com		Secret2.com	Secret2.com
			Public2.com			Public2.com
			Other			Default.com
		Other	Public2.com		Public2.com	Public2.com
			Other			Default.com
	Secret2.com	-	-	Default.com	Unexpected message	
	Other	-	-	Default.com	Unexpected message	

or *public2.com* only when the corresponding SNI is set. In other cases, the client will always get the certificate of *default.com*. Secret websites can never be accessed directly with the original protocol: they can only be accessed with SNIT. The server will behave as if the secret websites do not exist. The client can only reach the secret websites if the correct SNI, password and HTTP request are provided.

The differentiation between the original protocol and SNIT occurs in ClientFinished. Cases with wrong passwords are not listed in the table: the server will always terminate the connection with an **unexpected_message** error. Different passwords are allowed for different websites in our implementation. The public SNI must be a valid public website for the use of SNIT. To access a secret website, the client must first receive the correct certificate. For example, to get the certificate of *secret1.com*, the public SNI and the secret SNI must be set as *public1.com* and *secret1.com*, respectively. As the table shows, the server will only repeat the certificate of *public1.com* if the secret SNI is wrong. In this implementation, to reach *secret1.com*, the hostname in the HTTP request must be *secret1.com*. Different pairs of public and secret websites do not interfere with each other. The user cannot access *secret1.com* with a public SNI of *public2.com*. In practical applications, this constraint can be waived for convenience.

In this chapter, the implementation of SNIT is described in detail. The modification to the source code of Mbed TLS and the behaviours of Hiawatha webserver are explained. This implementation provides the security properties discussed in the next chapter.

Chapter 6

Security evaluation

Security is the most crucial part of TLS and SNIT. Our target is to disguise a secret website as a public website and minimize the differences that can be observed by the censor. A detailed analysis will be given in this chapter.

6.1 Possible blocking methods

SNIT is a modified version of TLS. The secret website is disguised as one or more public websites. Once the censor suspects that a public website is using this protocol, it can block the public website with the methods effective for websites using HTTPS, and the secret website can no longer be accessed through the blocked public website. These methods include DNS manipulation, certificate filtering, SNI blocking, and IP blocking. However, their shortcomings still exist. DNS manipulation can be bypassed by DNS over HTTPS (DoH), and SNIT can also protect DoH. If this protocol is broadly adopted, blocking will be a cat-and-mouse game.

SNI blocking is better in comparison, but it has the same premise to apply: the

censor must suspect the public website first. However, there is an exception: the SNI whitelist. In this approach, the censor will block all TLS connections whose SNI is not on the whitelist. Unfortunately, there is no perfect solution for this. Under the SNI whitelist, the public SNI must be set as an allowed website. The server must own and host the allowed website, or this is prone to active probing. However, only fully compliant websites can be added to the whitelist. Their owners are less likely to allow a secret service to exist. However, the SNI whitelist is too strict: all normal websites which are not on the whitelist are all blocked. Even in China, only a small-scale pilot has been conducted [52].

6.2 Passive probing

The most important security consideration is the concealment of SNIT. When a user accesses a secret website under SNIT, all the unencrypted parts of the session are identical to the respective public website. Figures 6.1 and 6.2 show the comparison of the two protocols.

The client only requests a single web page from the server in all sessions. The packets sent by the client are marked as blue, while those sent by the server are marked as orange. For both versions, the SNI in ClientHello is *public1.com*, and the ServerCertificate belongs to *public1.com*. From the censor's point of view, the only difference SNIT brings is the three additional encrypted packets. Since no changes have been made to the process in the key exchange, the difficulty of cracking the master secret or ciphers has not decreased.

Even so, the locations and sizes of these three packets are relatively fixed, so

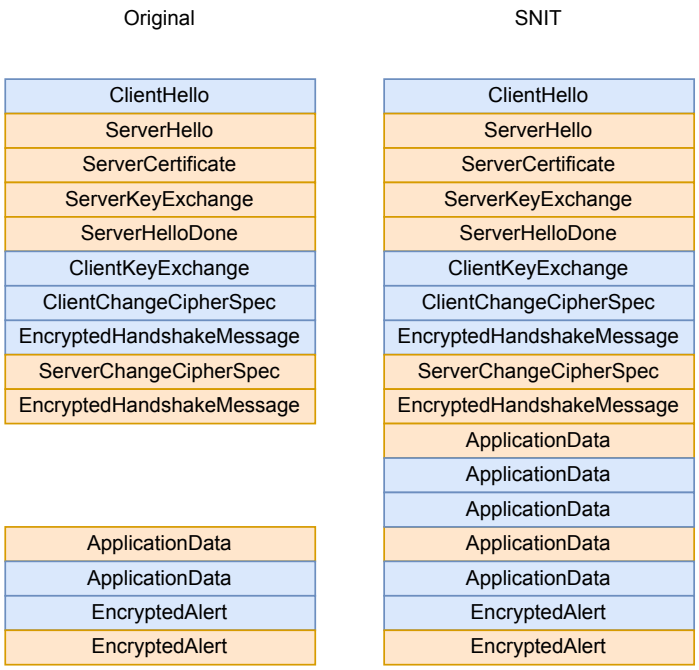

Figure 6.1: The packets sent in an HTTP session of the original TLS 1.2 handshake protocol and SNIT

the traffic characteristics of the two websites are not identical. Once the difference is significant enough for the censor to suspect that the user is accessing another website, not the corresponding public SNI, it will get blocked. Such an attack is feasible in theory [61, 62]. For a long time, the GFW has used traffic analysis to block censorship circumvention tools [63], not websites. Whether it will be used on websites in the future is unknown. Traffic shaping can be applied. We leave this for future work (see Chapter 9).

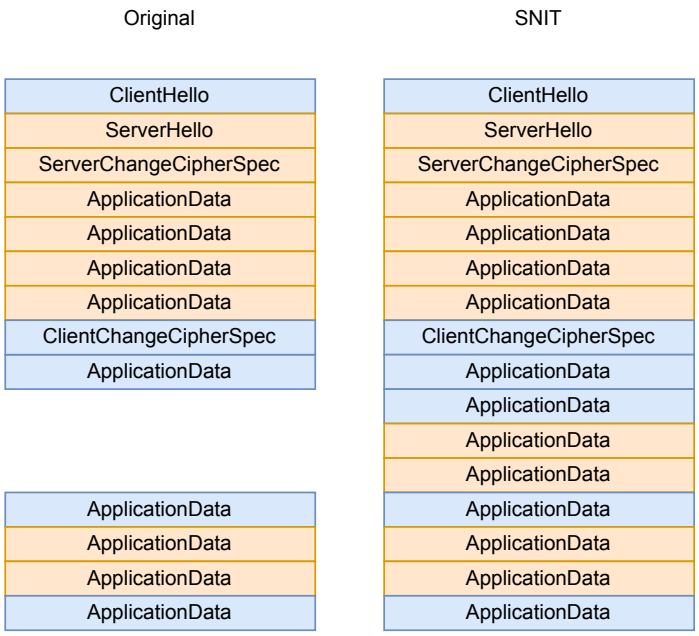

Figure 6.2: The packets sent in an HTTP session of the original TLS 1.3 handshake protocol and SNIT

6.3 Active probing

Active probing is a more severe threat: the censor can disguise itself as the user, try to establish a connection with the server, and check if there is any abnormality. The security goal is one can only identify the modified server if it holds the correct public and secret SNI and password. No matter which one is incorrect, the modified server performs like an unmodified one.

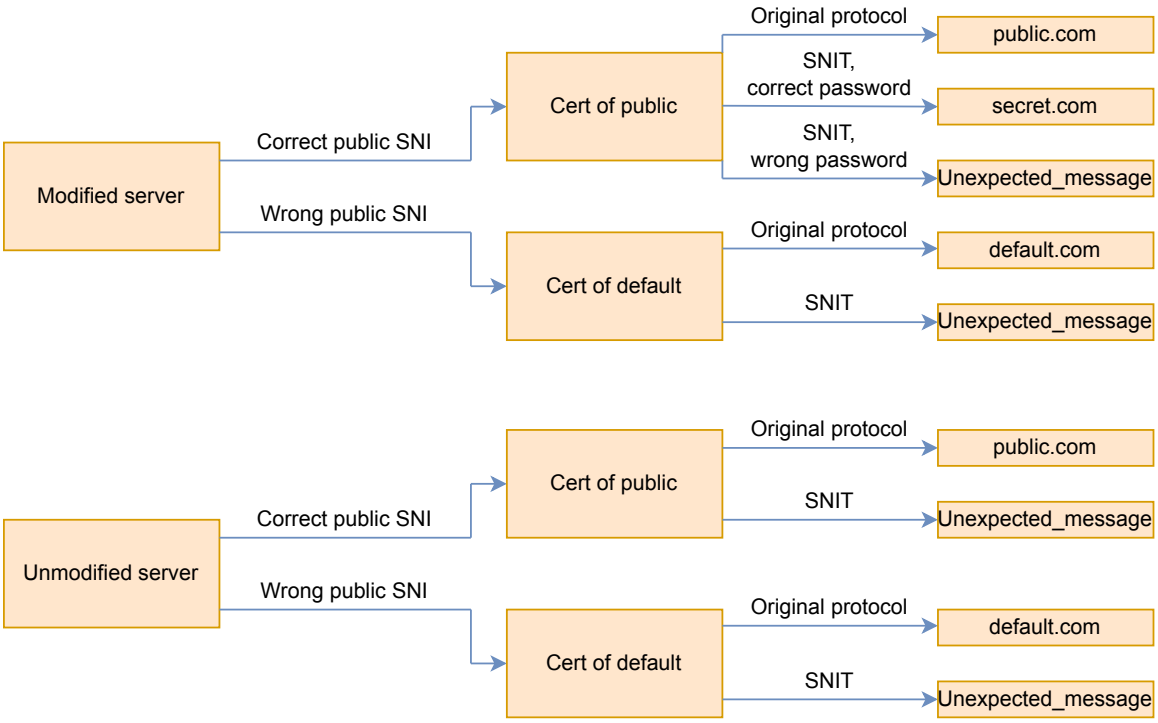

Figure 6.3: Comparison of the behaviours of a modified and an unmodified server when the client provides corresponding information

Figure 6.3 shows the server's behaviours when the client (or censor) provides different information. They simulate the situation when the censor performs an active probing without specific information, public SNI, secret SNI or password. The server will always send a public certificate, and if SNIT is applied, it will also send a secret certificate or terminate the connection with an unexpected_message error. The upper half of Figure 6.3 refers to the server hosting a public and a secret website with SNIT. In contrast, the lower half of Figure 6.3 refers to an unmodified server only hosting a public website. Although different implementations of webservers may behave differently, we take the Hiawatha webserver as an example.

No secret SNI or password is required if the client uses the original protocol. Both servers will reply with the default certificate if the client does not provide the correct public SNI or the public certificate if on the contrary. Also, the censor will always get an unexpected_message error from both servers with SNIT once it does not provide the correct password. SNIT replaces the handshake type **finished** (0x14) with **fake_finished** (0x15). An unmodified server cannot identify 0x15, resulting in an **unexpected_message** error. For the server using SNIT, as long as the message type is set as 0x15 and the **verify_data** (including password) is incorrect, it will always return such an error. This is exactly the behaviour of an unmodified server. If no password is set, the server will proceed after receiving the ClientFinished with 0x15. Though the following steps are different under TLS 1.2 or TLS 1.3, the client will never receive the **unexpected_message** and thus confirm that the server is using SNIT.

The public SNI is also an essential part of SNIT. As introduced in Chapter 4, it must be distributed along with the password. First, the server must host the domain of the public SNI. The censor can intercept the unencrypted public SNI in the ClientHello message and try to establish a TLS connection with the public SNI. In practical applications, if the server is not hosting the corresponding domain, the server will either send an error message or the default certificate, just like our implementation of Hiawatha. The censor learns that users can establish a TLS connection with a nonexistent SNI, and the server must be protecting a secret service. This problem also exists for all similar approaches, including HTTPT and BlindTLS (see Chapter 8). Another approach is proposed in [2]. The server generates a self-signed certificate each time a client requests a domain that the server is not hosting. Since self-signed

certificates are widely used among developers, blocking all self-signed certificates indiscriminately is inappropriate. However, the censor may request many random server names from the server. If it can always get a new self-signed certificate, the censor can block its IP address.

In addition to SNIT, the secret website can be set as accessible by users outside of the perimeter of the censored area using the original protocol at the same time. This provides convenience for those users since there is no need for additional distribution. If only it uses a different IP address from any public website, the safety of the public website will not be reduced. Moreover, repeated IP addresses should be avoided for any public websites. If IP blocking is applied by the censor, multiple public websites will be blocked together if they share the same IP address.

The certificate of the public website should also be verified by the client, though it is not the website that the client really accesses. Without verification, the public website might be forged by the censor. The secret SNI and password will then fall into the censor's hands. In conclusion, active probing cannot pose a threat if the server and the client meet the above requirements.

This chapter contains the security evaluation of SNIT. Possible blocking methods are analyzed, and passive probing and active probing are demonstrated to be ineffective against SNIT. In the next chapter, we describe the performance evaluation of SNIT.

Chapter 7

Performance evaluation

Since additional packets are sent during handshake, SNIT imposes a performance cost. On the other hand, application data is not modified. If the proportion of handshake is small enough, the performance loss can be negligible. In this part, the performance of the original protocol and SNIT has been tested and compared. The tests are carried out in LAN, and the latency is negligible. The time from the connection establishment to receiving the HTTP response is used as the standard for performance. Each session is repeated 1,000 times. The average, 10th percentile and 90th percentile values are reported.

Figure 7.1 shows the gap between the performance of the original protocol and SNIT. The client is the customized client demonstration program in the new library since it has to support SNIT. To reflect the speed of the handshake, application-level HTTP responses are minimized. Generally, SNIT takes an additional 30-40 milliseconds, which results in an average performance loss of 37.2% or 31.69 ms. Only the additional roundtrip and certification verification significantly affect performance.

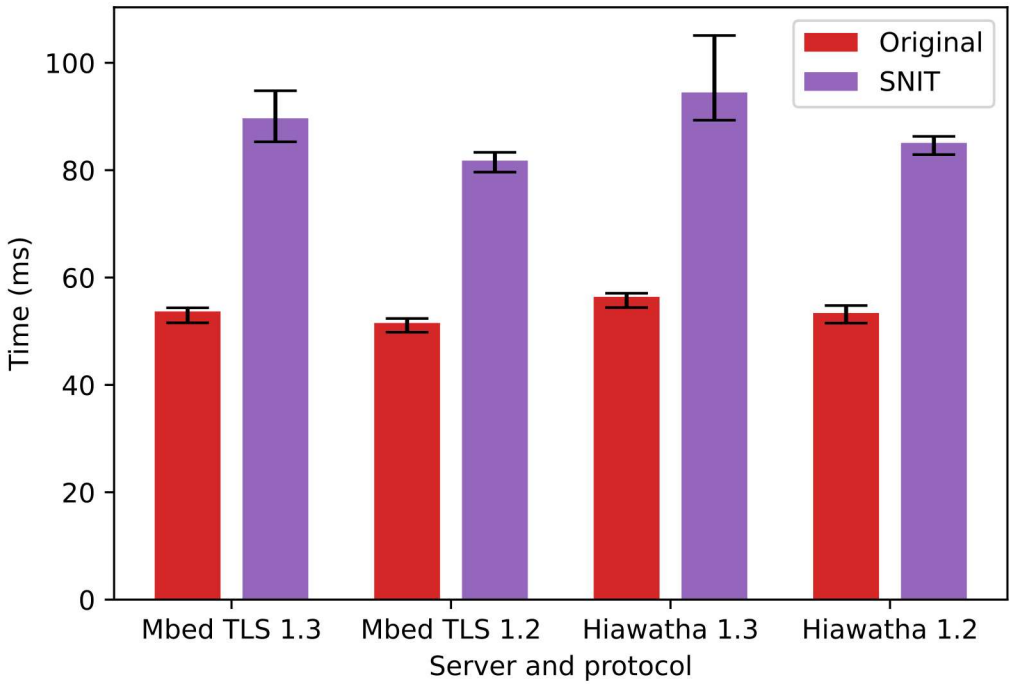

Figure 7.1: Time required to complete a connection for each protocol and server

In real applications, these costs will be amortized over the complete session, which is shown in figure 7.2. Due to its complexity, the Hiawatha webserver is slightly slower than the simple server program in the Mbed TLS library. Although TLS 1.3 is slightly slower than TLS 1.2 in this test, the saved round trip will turn it around in practical applications but not in a local area network.

Figure 7.2 shows the performance of both protocols for larger HTTP responses from 512 KB to 64 MB. As the size increases, the performance loss is increasingly amortized. The performance gap at 64 MB is only 2%. In practical applications, the client can access many web resources during a session, and the handshake does not need to be performed again. The longer latency also makes the performance loss even

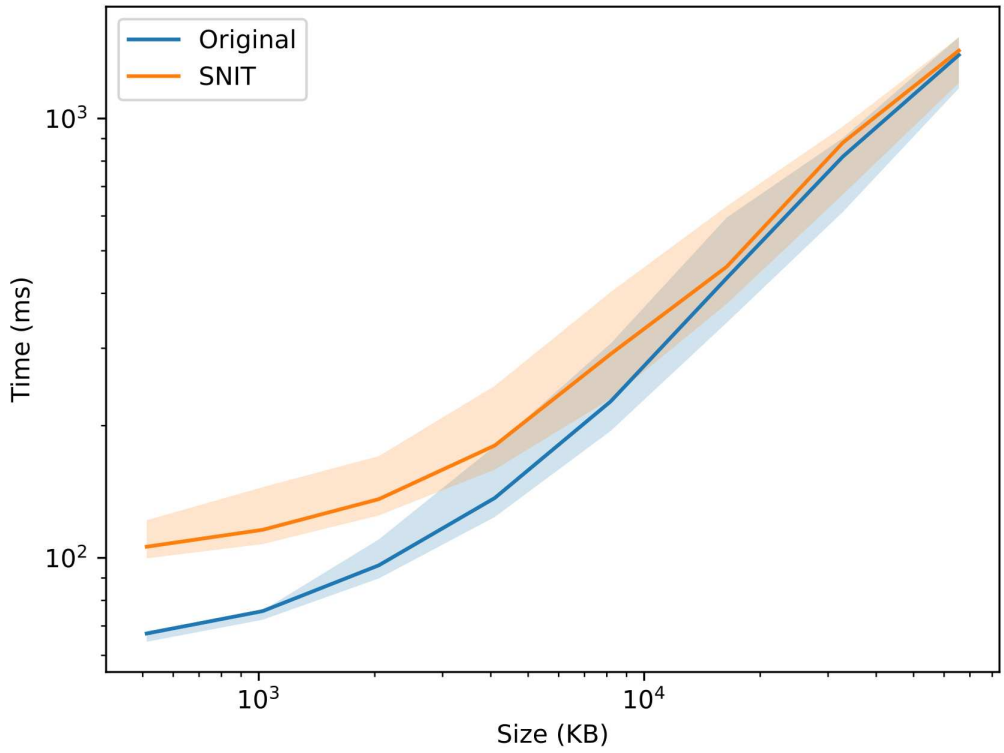

Figure 7.2: Time required to get different sizes of HTTP response

more negligible.

Figure 7.3 illustrates the performance of the unmodified handshake protocol. The TLS version and server program are fixed in each group, while the server and client programs in the original library and our library for implementation are compared. The results within each group are very close. This means the implementation of SNIT has very little performance impact on the original protocol. Users only accessing the public websites pay almost nothing for the application of SNIT.

In this chapter, the performance evaluation of SNIT is performed. Under all circumstances, the application of SNIT does not cause significant performance loss.

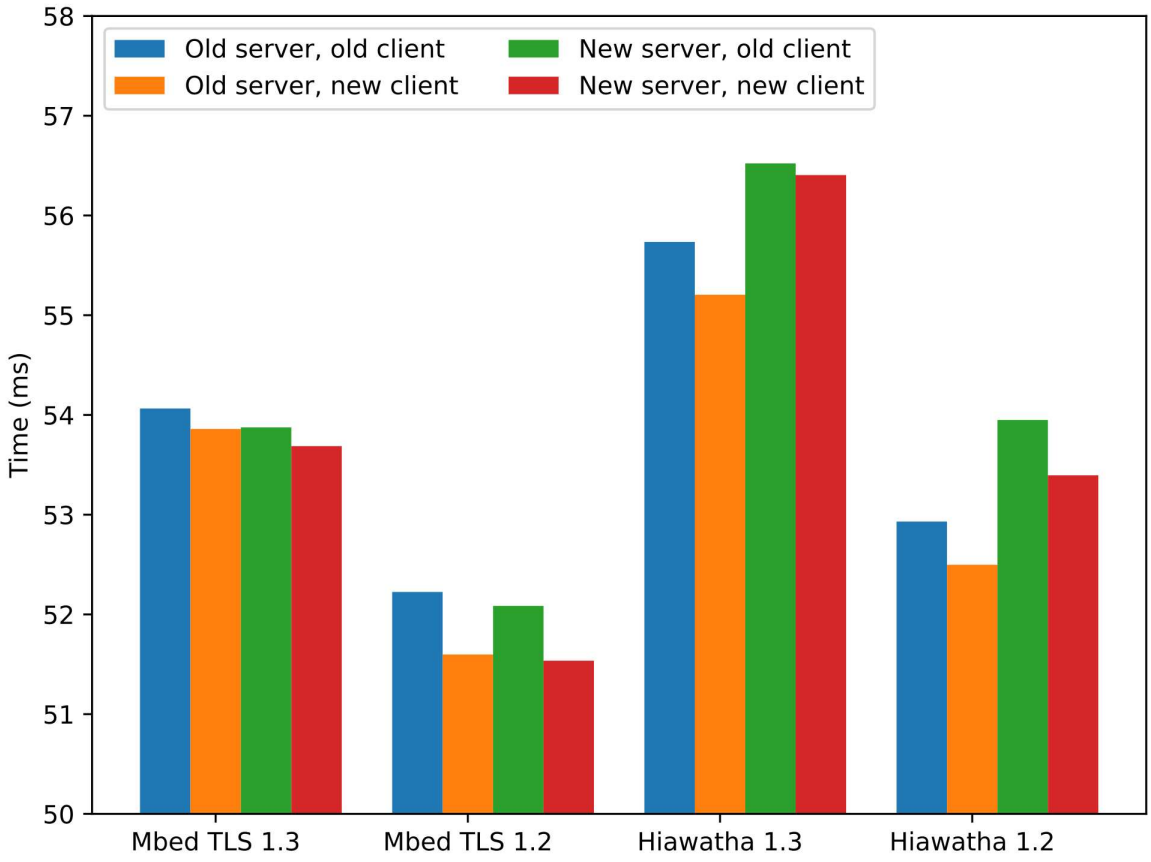

Figure 7.3: Time required to perform an original handshake with different versions of server and client

Chapter 8

Comparison to related work

Many solutions have been proposed to encrypt or spoof the unencrypted SNI. They will be introduced in detail and compared with SNIT.

8.1 ESNI and ECH

In TLS 1.3, the standard that has been adopted so far is Encrypted-SNI (ESNI) [64] and its upgraded version, Encrypted ClientHello (ECH) [65]. Until now, only a few websites support ESNI or ECH [66, 67].

ESNI extension type	ESNI extension length	ESNI extension body		
		Cipher suite	Key share length	Key share
0xFFCE	2 bytes	2 bytes	2 bytes	Key share len bytes

ESNI extension body			
Record digest length	Record digest	Encrypted SNI length	Encrypted SNI
2 bytes	Record digest length bytes	2 bytes	Encrypted SNI length bytes

Figure 8.1: The message format of the ESNI extension

The process of ESNI is as follows. An additional public key is added to the DNS records of the server's domain name. The server keeps the corresponding private key. When a client makes a DNS query, it will receive the server's IP address and the public key. This public key plays the same role as the public key in the key exchange. The client will perform a key exchange with the public key. It is not the key exchange in the handshake and is only used to protect the SNI. It puts the required cryptographic information and the encrypted server name in the ESNI extension. The ESNI extension belongs to ClientHello and replaces the SNI extension. The format of ESNI is shown in Figure 8.1. The cipher suite and key share are used for the key exchange. The record digest is generated by the hash function in the cipher suite for data integrity. The SNI is encrypted and placed in the encrypted SNI part of the extension. Upon receiving the ClientHello, the server finishes the key exchange and decrypts the server name. The confidentiality of ESNI is as good as that of the application data.

However, although it cannot be broken, its use can be observed by the censor. The extension type, 0xFFCE, is not encrypted, so the censor may block all ClientHello messages with a 0xFFCE extension and force the users to switch to the original SNI extensions. This has been implemented by the Great Firewall [68].

ECH is the evolved version of ESNI. It replaces the ESNI extension with a new extension called ClientHelloInner (CHI), which includes a standard ClientHello structure. This allows all the information a ClientHello includes to be encrypted, not only the SNI. It has the same disadvantage: the extension type can be easily identified. There is a new standard feature of ECH, called GREASE [65], in which clients that support ECH always send an ECH extension in every connection, whether the server

supports ECH or not. This would make selective blocking of ECH connections impossible. However, ECH is not a mandatory standard. All clients supporting ECH must support the regular ClientHello, and the censor can still block all connections with ECH. This forces all clients to use the regular ClientHello. The reason why the GFW has yet to block ECH may be because ECH is less popular than ESNI [69].

8.2 Other studies

Some other tools have been developed for SNI-based censorship circumvention. In [70], a design based on early data is introduced. In TLS 1.3, early data is an extension in the ClientHello. It is encrypted by the pre-shared key, which has to be obtained by the client from an earlier session. The original intention is to reduce latency and improve performance. In this design, a ClientHello#2 with a hidden SNI is placed in the early data, and the outer ClientHello includes a public SNI, which is only used to confuse the censor. After receiving the ClientHello, the server decrypts the early data and proceeds with the hidden SNI. This design has a number of flaws, which have been addressed in [70] and [64]. First, the server may be unable to distinguish the inner ClientHello from application data. Second, early data may not be supported, especially when the server is under DDoS. Also, early data is prone to replay attacks [71]. Although some studies are improving its security [72, 73], whether the replay attacks can be completely prevented remains a question.

Domain Fronting [54] is one approach deployed on CDN servers. Although CDN servers have to check the SNI to send the corresponding certificate, the host is chosen by the hostname in the HTTP request. For example, the client can establish a TLS

connection with the SNI of *public.com*, get the certificate of *public.com*, and then request *secret.com*. However, the client does not get the certificate of *secret.com*, so a fronting server spoofing attack can be applied: the attacker distributes a fake public website and misleads the users to a malicious secret website. As a result, major CDNs have blocked Domain Fronting [74, 75]. An additional certificate verification or even HTTPS-over-TLS could be added on the application layer, but this would cause performance degradation.

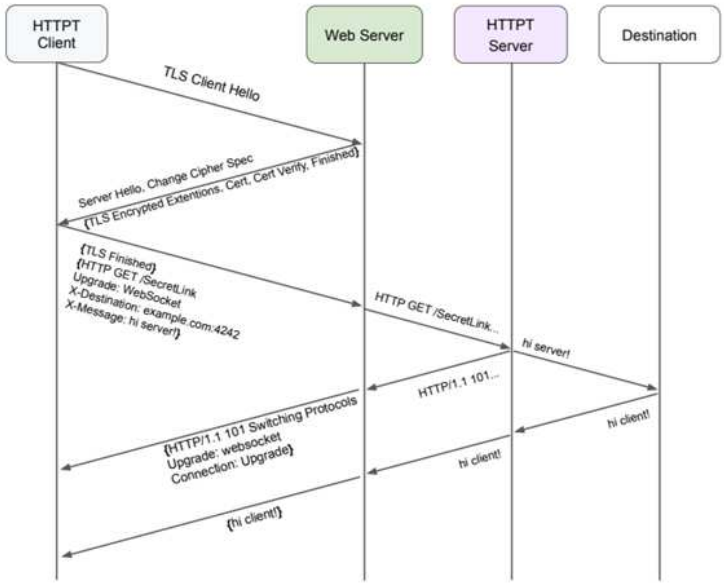

Figure 8.2: HTTPT handshake [1]

HTTPT [1] takes a similar approach. It functions as a TLS server, which also has secret proxy functionality, as shown in Figure 8.2. First, the HTTPT client performs a TLS handshake with the web server. The web server forwards all application data to the HTTPT server. Then, in application data, if the client sends a secret URL

specified by the HTTPT server, the HTTPT server will redirect to the secret destination. The secret link must be distributed out-of-band. The secret destination can be an HTTP server or any TCP server. If the client does not know the secret link, the HTTPT server will perform as a regular HTTP server, which serves an innocuous website. This resists the active probing attacks. HTTPT has the same disadvantages as domain fronting: it is prone to fronting server spoofing attacks, and additional verification will reduce performance.

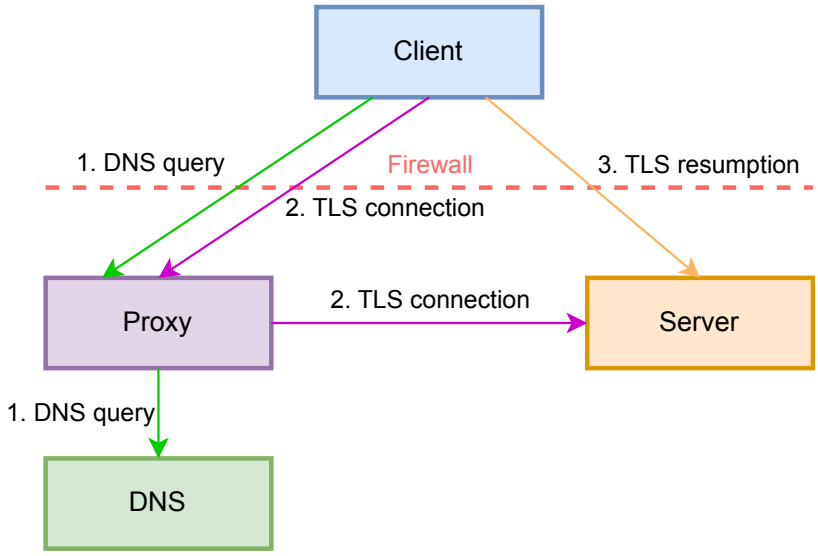

Figure 8.3: The flow of BlindTLS [2]

BlindTLS [2] takes another approach. It is based on session resumption, a feature defined in both TLS 1.2 and TLS 1.3. A previous session can be resumed once a client provides the session ID, and the handshake steps between ServerHello and ChangeCipherSpec are skipped. This improves performance. When using BlindTLS, the client needs a bridge. First, the client sends a DNS query to get the server IP address of the blocked website, which is still named *secret.com* and performs the

TLS handshake with the server. The SNI is *secret.com*. Both must be done via the bridge. When the server sends back a resumption ticket via the bridge, the client resumes the session without the bridge with the server. This time, the SNI is set as an innocuous website like *public.com*. Then, both sides can communicate normally. This method has a number of significant disadvantages. First, a bridge is necessary to initiate a session. This causes a serious performance loss. Second, the censor may drop all TLS session resumption packets. This method will block BlindTLS, while innocent users only suffer some performance penalty [2]. This prevents BlindTLS from massive deployment. Last, in TLS 1.3, the SNI in the resumption packet must match the one sent in the ClientHello in the initial handshake. Unless this rule can be broken, BlindTLS cannot work for TLS 1.3.

8.3 Comparison of approaches

Table 8.1: The comparison of SNI protection designs

	SNIT	HTTPT	DF	Early data	ESNI/ECH	BlindTLS
Additional roundtrip	Y	Y				
Out-of-band key/session	Y	Y		Y		Y
Out-of-band SNI	Y	Y	Y	Y		Y
TLS 1.2 supported	Y	Y	Y			Y
TLS 1.3 supported	Y	Y	Y	Y	Y	
Gateway authentication	Y	Y	Y		N/A	N/A
Secret authentication	Y			Y	Y	Y
Bridge required						Y
Characteristics in plaintext					Y	
Early data required				Y		

The properties of all approaches mentioned in this Chapter are shown in Table 8.1. All current designs have shortcomings. Though ESNI and ECH have become standard with excellent performance and convenience, they have distinctive characteristics in plaintext. Unless ECH becomes a mandatory standard, the censor can block it indiscriminately. The disadvantages of the early data approach and BlindTLS have been introduced in Section 8.2. These two designs are almost impossible to apply practically. HTTPT and Domain Fronting are similar. They are prone to fronting server spoofing attacks: the attacker distributes a fake public website and misleads the users to a malicious secret website. This can be mitigated by compulsory certificate verification in the application layer. In principle, with the additional verification,

they are almost identical to SNIT. Since additional application-layer interaction is not recommended [64], this is an advantage of SNIT.

In this chapter, many related works are introduced and compared with SNIT. Compared to competitive approaches, SNIT has decent overall security and performance.

Chapter 9

Future work

SNIT's security and performance meet expectations, but there is room for further improvement. Traffic analysis still poses a threat since the traffic characteristics of the public and secret websites are not identical. Some additional measures, e.g. manually adding delay and padding, could be taken to minimize the impact of the three additional messages. The difference in traffic characteristics could also be minimized with traffic shaping [76, 77, 78], though this should be implemented in the application layer above SNIT. Also, to deploy SNIT in practical applications like the solution of reverse proxy and CDN, more features on the Hiawatha webserver and other tools are to be further developed. The need for out-of-band distribution is the biggest disadvantage of SNIT and most similar approaches. Further research on distribution is also important.

Chapter 10

Conclusion

Internet censorship is a global problem. Many countries censor the internet for different reasons. This threatens internet freedom and access to information. Although the application of TLS significantly improves security, its weaknesses can still be exploited for internet censorship. The unencrypted SNI directly reveal the website's identity. In this research, we proposed a modified handshake protocol, SNIT, for both TLS 1.2 and TLS 1.3. It replaces the unencrypted SNI with an innocuous one and tunnels the secret SNI after the normal handshake process. The censor cannot learn the existence of the secret website unless it acquires the password. The protocol is also highly resistant to passive and active probing. Its compatibility with regular TLS is maintained: the server can host multiple public and secret websites with regular TLS and SNIT. The largest disadvantage of SNIT is the out-of-band distribution, though this is common for similar approaches. It adds an average of 31.69 ms to the initial TLS handshake, and there is no effect on subsequent traffic. Compared to competitive approaches, SNIT has decent overall security and performance. Despite

this, no current approach is completely satisfactory. The default unencrypted SNI extension is the biggest enemy of anti-censorship. To achieve perfect secrecy, the SNI must be spoofed. In conclusion, TLS is only a part of internet censorship. Perfect secrecy can never be achieved with one protocol only.

Bibliography

[1] S. Frolov and E. Wustrow, "HTTPT: A Probe-Resistant Proxy," in *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*, 2020.

[2] S. Satija and R. Chatterjee, "BlindTLS: Circumventing TLS-based HTTPS censorship," in *Proceedings of the ACM SIGCOMM 2021 Workshop on Free and Open Communications on the Internet*, pp. 43–49, 2021. doi: 10.1145/3473604.3474564.

[3] S. Chandel, Z. Jingji, Y. Yunnan, S. Jingyao, and Z. Zhipeng, "The golden shield project of china: A decade later—an in-depth study of the great firewall," in *2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pp. 111–119, IEEE, 2019. doi: 10.1109/CyberC.2019.00027.

[4] "What Is Censorship? — American Civil Liberties Union — aclu.org." https://www.aclu.org/documents/what-censorship, 2006. [Accessed 27-03-2024].

[5] S. Aryan, H. Aryan, and J. A. Halderman, "Internet censorship in iran: A first look," in *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI 13)*, 2013.

[6] M. Akgül and M. Kırlıdoğ, "Internet censorship in turkey," *Internet Policy Review*, vol. 4, no. 2, pp. 1–22, 2015. doi: 10.14763/2015.2.366.

[7] V. Agrawal and P. Sharma, "Internet censorship in india," in *Proceedings of 10th International Conference on Digital Strategies for Organizational Success*, 2019. doi: 10.2139/ssrn.3309268.

[8] S. A. Meserve and D. Pemstein, "Terrorism and internet censorship," *Journal of peace research*, vol. 57, no. 6, pp. 752–763, 2020. doi: 10.1177/0022343320959369.

[9] A. Shishkina and L. Issaev, "Internet censorship in Arab countries: Religious and moral aspects," *Religions*, vol. 9, no. 11, p. 358, 2018. doi: 10.3390/rel9110358.

[10] "Usage statistics of default protocol https for websites, may 2023 — w3techs.com." https://w3techs.com/technologies/details/ce-httpsdefault. [Accessed 28-05-2023].

[11] P. Hoffman and P. McManus, "DNS queries over HTTPS (DoH)," tech. rep., 2018. doi: 10.17487/rfc8484.

[12] R. Chhabra, P. Murley, D. Kumar, M. Bailey, and G. Wang, "Measuring DNS-over-HTTPS Performance around the World," in *Proceedings of the 21st ACM Internet Measurement Conference*, pp. 351–365, 2021. doi: 10.1145/3487552.3487849.

[13] L. Jin, S. Hao, H. Wang, and C. Cotton, "Understanding the impact of encrypted DNS on internet censorship," in *Proceedings of the Web Conference 2021*, pp. 484–495, 2021. doi: 10.1145/3442381.3450084.

[14] K. Bock, Y. Fax, K. Reese, J. Singh, and D. Levin, "Detecting and evading Censorship-in-Depth: A case study of Iran's protocol whitelister," in *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*, 2020.

[15] T. K. Yadav, A. Sinha, D. Gosain, P. K. Sharma, and S. Chakravarty, "Where the light gets in: Analyzing web censorship mechanisms in india," in *Proceedings of the Internet Measurement Conference 2018*, pp. 252–264, 2018. doi: 10.1145/3278532.3278555.

[16] J. L. Hall, M. D. Aaron, S. Adams, A. Andersdotter, B. Jones, and N. Feamster, "A survey of worldwide censorship techniques," tech. rep., Internet-Draft draft-irtf-pearg-censorship-04. Internet Engineering Task ..., 2020.

[17] K. Bock, G. Naval, K. Reese, and D. Levin, "Even censors have a backup: Examining china's double https censorship middleboxes," in *Proceedings of the ACM SIGCOMM 2021 Workshop on Free and Open Communications on the Internet*, pp. 1–7, 2021. doi: 10.1145/3473604.3474559.

[18] D. Xue, B. Mixon-Baca, ValdikSS, A. Ablove, B. Kujath, J. R. Crandall, and R. Ensafi, "TSPU: Russia's decentralized censorship system," in *Proceedings of the 22nd ACM Internet Measurement Conference*, pp. 179–194, 2022. doi: 10.1145/3517745.3561461.

[19] K. Elmenhorst, B. Schütz, N. Aschenbruck, and S. Basso, "Web censorship measurements of HTTP/3 over QUIC," in *Proceedings of the 21st ACM Internet Measurement Conference*, pp. 276–282, 2021. doi: 10.1145/3487552.3487836.

[20] R. Dingledine, N. Mathewson, P. F. Syverson, *et al.*, "Tor: The second-generation onion router.," in *USENIX security symposium*, vol. 4, pp. 303–320, 2004. doi: 10.21236/ada465464.

[21] "The Tor Project — Privacy & Freedom Online — torproject.org." https://www.torproject.org/about/sponsors/. [Accessed 28-03-2024].

[22] P. Winter and J. R. Crandall, "The Great Firewall of China: How it blocks Tor and why it is hard to pinpoint," *Login: The Usenix Magazine*, vol. 37, no. 6, pp. 42–50, 2012.

[23] A. Dunna, C. O'Brien, and P. Gill, "Analyzing china's blocking of unpublished tor bridges," in *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI 18)*, 2018.

[24] S. J. Murdoch and G. Kadianakis, "Pluggable transports roadmap," *The Tor Project, Tech. Rep*, pp. 03–003, 2012.

[25] Q. Tan, X. Wang, W. Shi, J. Tang, and Z. Tian, "An anonymity vulnerability in Tor," *IEEE/ACM Transactions on Networking*, vol. 30, no. 6, pp. 2574–2587, 2022. doi: 10.1109/tnet.2022.3174003.

[26] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal, "RAPTOR: Routing attacks on privacy in tor," in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 271–286, 2015.

[27] H. Al Jawaheri, M. Al Sabah, Y. Boshmaf, and A. Erbad, "Deanonymizing Tor hidden service users through Bitcoin transactions analysis," *Computers & Security*, vol. 89, p. 101684, 2020. doi: 10.1016/j.cose.2019.101684.

[28] X. Zhang, X. Ma, X. Han, B. She, and W. Li, "An uncertainty-based traffic training approach to efficiently identifying encrypted proxies," in *2020 12th International Conference on Advanced Infocomm Technology (ICAIT)*, pp. 95–99, IEEE, 2020. doi: 10.1109/icait51223.2020.9315573.

[29] Y. Wang, G. Yu, W. Shen, and L. Sun, "Deep learning based on byte sample entropy for VPN encrypted traffic identification," in *2022 5th International Conference on Advanced Electronic Materials, Computers and Software Engineering (AEMCSE)*, pp. 293–296, IEEE, 2022. doi: 10.1109/aemcse55572.2022.00066.

[30] H. Wu, Y. Liu, G. Cheng, and X. Hu, "Real-time identification of VPN traffic based on counting Bloom filter and chained hash table from sampled data in high-speed networks," in *ICC 2022-IEEE International Conference on Communications*, pp. 5070–5075, IEEE, 2022. doi: 10.1109/icc45855.2022.9839256.

[31] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2," tech. rep., 2008. doi: 10.17487/rfc5246.

[32] E. Rescorla, H. Tschofenig, and N. Modadugu, "The datagram transport layer security (DTLS) protocol version 1.3," in *RFC 9147*, 2022. doi: 10.17487/rfc9147.

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

[34] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," tech. rep., 2018. doi: 10.17487/rfc8446.

[35] "Qualys SSL Labs - SSL Pulse — ssllabs.com." https://www.ssllabs.com/ssl-pulse/. [Accessed 28-03-2024].

[36] S. Malladi, J. Alves-Foss, and R. B. Heckendorn, "On preventing replay attacks on security protocols," in *Proc. International Conference on Security and Management*, vol. 6, Citeseer, 2002. doi: 10.1109/CCST.2011.6095943.

[37] H. Krawczyk, M. Bellare, and R. Canetti, "RFC2104: HMAC: Keyed-hashing for message authentication," 1997. doi: 10.17487/rfc2104.

[38] M. Bellare, R. Canetti, and H. Krawczyk, "Message authentication using hash functions: The HMAC construction," *RSA Laboratories' CryptoBytes*, vol. 2, no. 1, pp. 12–15, 1996. doi: 10.1145/141809.141812.

[39] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pp. 1–15, Springer, 1996. doi: 10.1007/3-540-68697-5_1.

[40] B. Preneel and P. C. Van Oorschot, "MDx-MAC and building fast MACs from hash functions," in *Annual International Cryptology Conference*, pp. 1–14, Springer, 1995. doi: 10.1007/3-540-44750-4_1.

[41] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright, "RFC3546: Transport Layer Security (TLS) Extensions," 2003. doi: 10.17487/rfc3546.

[42] M. Ray and S. Dispensa, "Renegotiating tls," 2009.

[43] J. Orchilles, "TLS/SSL Renegotiation DoS. Internet Engineering Task Force," 2006.

[44] S. Hollenbeck, "Transport layer security protocol compression methods," tech. rep., 2004. doi: 10.17487/rfc374.

[45] C. Fournet and M. Kohlweiss, "TLS Compression Fingerprinting and a Privacy-aware API for TLS," 2012.

[46] L. Foppe, J. Martin, T. Mayberry, E. C. Rye, and L. Brown, "Exploiting tls client authentication for widespread user tracking," *Proceedings on Privacy Enhancing Technologies*, 2018. doi: 10.1515/popets-2018-0031.

[47] D. Eastlake 3rd, "RFC 6066: Transport Layer Security (TLS) Extensions: Extension Definitions," 2011. doi: 10.17487/rfc6066.

[48] K. Bock, G. Hughey, L.-H. Merino, T. Arya, D. Liscinsky, R. Pogosian, and D. Levin, "Come as you are: Helping unmodified clients bypass censorship with server-side evasion," in *Proceedings of the Annual conference of the ACM Special*

Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, pp. 586–598, 2020. doi: 10.1145/3387514.3405889.

[49] K. Bock, G. Hughey, X. Qiang, and D. Levin, "Geneva: Evolving censorship evasion strategies," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2199–2214, 2019. doi: 10.1145/3319535.3363189.

[50] Z. Wang, Y. Cao, Z. Qian, C. Song, and S. V. Krishnamurthy, "Your state is not mine: A closer look at evading stateful internet censorship," in *Proceedings of the 2017 Internet Measurement Conference*, pp. 114–127, 2017. doi: 10.1145/3131365.3131374.

[51] Z. Wang and S. Zhu, "SymTCP: Eluding stateful deep packet inspection with automated discrepancy discovery," in *Network and Distributed System Security Symposium (NDSS)*, 2020. doi: 10.14722/ndss.2020.24083.

[52] "Watch out for hidden mass downgrade attacks in SNI whitelisted areas · Issue #254 · net4people/bbs — github.com." https://github.com/net4people/bbs/issues/254, 2023. [Accessed 29-03-2024].

[53] K. Bock, P. Bharadwaj, J. Singh, and D. Levin, "Your censor is my censor: Weaponizing censorship infrastructure for availability attacks," in *2021 IEEE Security and Privacy Workshops (SPW)*, pp. 398–409, IEEE, 2021. doi: 10.1109/spw53761.2021.00059.

[54] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson, "Blocking-resistant communication through domain fronting," *Proceedings on Privacy Enhancing Technologies*, 2015. doi: 10.1515/popets-2015-0009.

[55] Q. Wang, Z. Lin, N. Borisov, and N. Hopper, "rBridge: User Reputation based Tor Bridge Distribution with Privacy Preservation.," in *NDSS*, 2013.

[56] F. Douglas, W. Pan, M. Caesar, *et al.*, "Salmon: Robust proxy distribution for censorship circumvention," *Proceedings on Privacy Enhancing Technologies*, 2016. doi: 10.1515/popets-2016-0026.

[57] L. Tulloch and I. Goldberg, "Lox: Protecting the social graph in bridge distribution," Master's thesis, University of Waterloo, 2022. doi: 10.56553/popets-2023-0029.

[58] "Mbed TLS." https://www.trustedfirmware.org/projects/mbed-tls, 2024. [Accessed 01-07-2024].

[59] "ESP-IDF." https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html, 2024. [Accessed 01-07-2024].

[60] "RIOT Documentation." https://api.riot-os.org/index.html, 2024. [Accessed 01-07-2024].

[61] X. Gong, N. Kiyavash, and N. Borisov, "Fingerprinting websites using remote traffic analysis," in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 684–686, 2010. doi: 10.1145/1866307.1866397.

[62] X. Gong, N. Borisov, N. Kiyavash, and N. Schear, "Website detection using remote traffic analysis," in *Privacy Enhancing Technologies: 12th International Symposium, PETS 2012, Vigo, Spain, July 11-13, 2012. Proceedings 12*, pp. 58–78, Springer, 2012. doi: 10.1007/978-3-642-31680-7_4.

[63] M. Wu, J. Sippe, D. Sivakumar, J. Burg, P. Anderson, X. Wang, K. Bock, A. Houmansadr, D. Levin, and E. Wustrow, "How the Great Firewall of China detects and blocks fully encrypted traffic," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 2653–2670, 2023.

[64] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood, "Encrypted server name indication for TLS 1.3," *IETF draft. Available at: https://tools. ietf. org/html/draft-ietf-tls-esni-02 (Accessed December 14th 2018)*, 2019.

[65] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood, "TLS Encrypted Client Hello," *IETF draft. Available at: https://tools. ietf. org/html/draft-ietf-tls-esni-17*, 2023.

[66] Y. Guan, W. Xia, G. Gou, P. Fu, B. Wang, and Z. Li, "Large-Scale Measurement of Encrypted TLS Server Name Indication (ESNI): How Far Have We Come?," in *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pp. 319–326, IEEE, 2021. doi: 10.1109/hpcc-dss-smartcity-dependsys53884.2021.00067.

[67] Z. Chai, A. Ghafari, and A. Houmansadr, "On the Importance of Encrypted-SNI (ESNI) to Censorship Circumvention," in *FOCI @ USENIX Security Symposium*, 2019.

[68] D. L. K. Bock, L. Merino, D. Fifield, A. Housmansadr, and D. Levin, "Exposing and Circumventing China's Censorship of ESNI," 2020.

[69] "Firefox will ship ECH by default · Issue #280 · net4people/bbs — github.com." https://github.com/net4people/bbs/issues/280, 2023. [Accessed 29-03-2024].

[70] C. Huitema and E. Rescorla, "SNI Encryption in TLS Through Tunneling," *IETF draft. Available at: https://tools. ietf. org/html/draft-ietf-tls-sni-encryption-00*, 2017.

[71] M. Fischlin and F. Günther, "Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 60–75, IEEE, 2017. doi: 10.1109/eurosp.2017.18.

[72] E. G. AbdAllah, R. Kuang, and C. Huang, "Generating just-in-time shared keys (JIT-SK) for TLS 1.3 zero RoundTrip time (0-RTT)," *Int. J. Mach. Learn. Comput.*, vol. 12, no. 3, pp. 96–101, 2022. doi: 10.18178/ijmlc.2022.12.3.1086.

[73] M. Abdelhafez, S. Ramadass, and M. Abdelwahab, "TLS Guard for TLS 1.3 zero round-trip time (0-RTT) in a distributed environment," *Journal of King Saud University-Computer and Information Sciences*, p. 101797, 2023. doi: 10.1016/j.jksuci.2023.101797.

[74] E. Doerr, "Securing our approach to domain fronting within Azure." https://www.microsoft.com/en-us/security/blog/2021/03/26/securing-our-approach-to-domain-fronting-within-azure, 2021. [Accessed 27-03-2024].

[75] C. MacCarthaigh, "Enhanced domain protections for amazon cloudfront requests." https://aws.amazon.com/blogs/security/enhanced-domain-protections-for-amazon-cloudfront-requests, 2018. [Accessed 27-03-2024].

[76] S. Xiong, A. D. Sarwate, and N. B. Mandayam, "Network traffic shaping for enhancing privacy in iot systems," *IEEE/ACM Transactions on Networking*, vol. 30, no. 3, pp. 1162–1177, 2022. doi: 10.1109/tnet.2021.3140174.

[77] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster, "Keeping the smart home private with smart (er) iot traffic shaping," *arXiv preprint arXiv:1812.00955*, 2018. doi: 10.2478/popets-2019-0040.

[78] C. Bocovich and I. Goldberg, "Slitheen: Perfectly imitated decoy routing through traffic replacement," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1702–1714, 2016. doi: 10.1145/2976749.2978312.