TÉCNICO
LISBOA

MultiTLS: Secure communication channel with diversity

Ricardo Manuel Mota de Moura

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Professor Miguel Filipe Leitão Pardal
Professor Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Professor Paolo Romano
Supervisor: Professor Miguel Filipe Leitão Pardal
Member of the Committee: Professor André Ventura da Cruz Marnôto Zúquete

June 2018

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First of all, I would like to thank my family, especially my parents for the examples and support they have always given me throughout my life, my brother for being my best friend and my girlfriend, Cristina, for the support and understanding in the most difficult moments. Secondly, I would like to thank my friends, especially those I met in college, highlighting the following: Rui Claro, Rafael Soares, Gonçalo Costa and Diogo Cruz.

I would like to express my gratitude and appreciation to my supervisors, Professor Miguel Pardal and Professor Miguel Correia for all the availability and sharing of your knowledge. Your help throughout the development of this work was fundamental and very relevant. I would also like to thank Professor Miguel Matos, David Matos and my fellow Masters' colleague, Isabel, for the help and suggestions given in the development of this project.

This work was supported by the European Commission through project H2020-653884 (Safe-Cloud) and by national funds through Fundacão para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

Resumo

A troca de mensagens com segurança é um requisito importante num sistema distribuído. Isto requer que os canais de comunicação tenham três propriedades principais: confidencialidade, integridade e autenticidade. O protocolo TLS fornece mecanismos que permitem criar canais de comunicação com essas propriedades. No entanto, as vulnerabilidades de projeto, de implementação e nos próprios mecanismos criptográficos podem tornar os canais de comunicação inseguros. Nesse caso, é aconselhável atualizar o software mas o processo de atualização pode ser demorado. Neste trabalho apresentamos o MultiTLS, um *middleware* baseado em mecanismos de diversidade e encapsulamento, que permite manter os canais de comunicação seguros mesmo quando uma nova vulnerabilidade é descoberta. O MultiTLS cria um canal de comunicação seguro através do encapsulamento de k canais TLS, em que cada um possui um conjunto de mecanismos criptográficos diferente dos restantes $k-1$. Esta abordagem permite que, mesmo que k–1 conjuntos de mecanismos criptográficos se tornem vulneráveis, o canal de comunicação permanece protegido devido ao restante conjunto de mecanismos criptográficos. A diversidade dos conjuntos de mecanismos criptográficos toleram falhas de criptográficas. Avaliou-se o desempenho do MultiTLS e comparou-se com o *Vulnerability-Tolerant Transport Layer Security* (vtTLS), um outro protocolo que fornece canais de comunicação seguros e tolerantes a vulnerabilidades com base na diversidade dos mecanismos criptográficos e dos certificados. Embora o vtTLS apresente melhor desempenho, conclui-se que o MultiTLS tem a vantagem de ser fácil de utilizar e de manter uma vez que não modifica nenhuma das suas dependências, tal como acontece com o vtTLS.

Palavras-chave: Canais de comunicação seguros, SSL/TLS, Segurança, Tolerância a vulnerabilidades, Diversidade para segurança, Encapsulamento

Abstract

Exchanging messages securely is an important requirement in a distributed system. This requires that communication channels have three main properties - confidentiality, integrity, and authenticity.

The TLS protocol provides mechanisms that allow creating communication channels with these properties. However, design, implementation, and cryptographic vulnerabilities can make communication channels insecure. In that case, it is advisable to update the software, but the update process may be time-consuming. In order to solve these problems, it is necessary to create mechanisms that allow the communication channels to be kept secure even when a new vulnerability is discovered.

In this work we present MultiTLS, a middleware based on diversity and tunneling mechanisms which allows keeping communication channels secure even when a new vulnerability is discovered. MultiTLS creates a secure communication channel through the encapsulation of k TLS channels, where each one has a different cipher suite compared to the remaining $k-1$. This approach allows, even when $k-1$ cipher suites become vulnerable, the communication channel remains protected due to the remaining cipher suite. The diversity of cipher suites tolerates cryptography faults. We evaluated the performance of MultiTLS and compared it with Vulnerability-Tolerant Transport Layer Security (vtTLS), another protocol that provides vulnerability-tolerant secure communication channels based on diversity and redundancy of cryptographic mechanisms and certificates. Although vtTLS performs better, we concluded that MultiTLS has the advantage of being easy to use and maintain since it does not modify any of its dependencies, as with vtTLS.

Keywords: Secure communication channels, SSL/TLS, Security, Vulnerability-tolerance, Diversity for security, Tunneling

Contents

Acknowledgments v

Resumo vii

Abstract ix

List of Tables xiii

List of Figures xv

1 Introduction **1**

1.1 Contributions 3

1.2 Outline 3

2 Background and Related Work **5**

2.1 SSL/TLS 5

2.1.1 The SSL/TLS protocol 5

2.1.2 TLS configuration variants 9

2.2 TLS Vulnerabilities 11

2.2.1 Design vulnerabilities 11

2.2.2 Implementation vulnerabilities 12

2.2.3 Cryptographic vulnerabilities 12

2.3 Achieving security through diversity 15

2.3.1 Automated software diversity 16

2.3.2 Vulnerability-Tolerant TLS 17

2.3.3 Tunneling 18

2.4 Summary 21

3 MultiTLS **23**

3.1 Design 24

3.2 Combining Diverse Cipher Suites 25

3.3 Running MultiTLS 28

3.4 Implementing the tunnels 29
3.5 Summary 31

4 Evaluation **33**
4.1 Performance costs 33
4.2 Comparisons with MultiTLS 36
4.3 Use case 38
4.4 Summary 39

5 Conclusions **41**
5.1 Future Work 41

Bibliography **43**

List of Tables

2.1 Key exchange algorithms supported by TLS. 10

2.2 Symmetric-key cipher algorithms supported by TLS. 10

2.3 Hash functions supported by TLS. 10

3.1 Cipher suites grouped by the combination of key exchange and authentication algorithms 26

3.2 Diversity metrics of MultiTLS symmetric cryptographic mechanisms 27

List of Figures

2.1 TLS protocol stack 6
2.2 TLS Record protocol procedure 6
2.3 Sequence diagram of the TLS Handshake protocol 7
2.4 Representation of a replicated service with n replicas 16
2.5 The format of the IP in IP encapsulation 20
2.6 Structure of the use of IPsec mechanisms, AH and ESP, in transport and tunnel modes 20
2.7 Representation of the SSH tunnel configuration 21

3.1 Example of using TUN interfaces 24
3.2 MultiTLS design with $k = 2$ and the flow of sending messages from one application to another on different hosts. 25
3.3 MultiTLS design with $k = 2$ to be used to establish communication between a browser and a proxy. 29

4.1 Comparison between the time it takes to send 1 MB, 100 MB and 1 GB messages in relation to the number of tunnels created. 34
4.2 The overhead of adding more tunnels in relation to k = 1. 35
4.3 Comparison between the time it takes to send 1 MB, 100 MB and 1 GB messages in relation to the number of unencrypted tunnels. 36
4.4 Overhead of the difference between first and second evaluation results. 37
4.5 Time for sending messages with 1MB, 100MB and 1GB in size via vtTLS, two MultiTLS tunnels and one DTLS communication over one MultiTLS tunnel. . . 38
4.6 MultiTLS design with $k = 2$ to be used to establish communication between a browser and a proxy. 39
4.7 Time to load sites used the following 4 approaches: no proxy, with proxy, with proxy using communication created by MultiTLS with a tunnel and with two tunnels. 40

Chapter 1

Introduction

We are currently living in an increasingly digital age, where a large part of the services, such as banking, shopping, healthcare, and voting, can be accessed through the Internet. Recently, there have been many cyber attacks that have caused increased losses and damages to businesses and Internet users [1]. Some of these attacks against the communication confidentiality have been performed by government-related organizations, such as the National Security Agency (NSA). They have collected multiple data from services of several companies [2]. Other attacks, such as WannaCry, are organized by groups of hackers who make money by restricting access to the infected system until the victims pay a ransom [3]. These cases show that nowadays, the use of secure communication protocols is a fundamental component of distributed systems and digital business because it allows entities to exchange messages through a secure communication channel on the Internet. These channels guarantee the following three properties:

- Confidentiality - this ensures that only the receiver is able to read the message;
- Integrity - it ensures that messages can not be changed without the receiver detecting it;
- Authenticity - it ensures that the identity is who it claims to be, this prevents third parties from impersonating the entities involved in the channel.

Transport Layer Security (TLS) is one of the most commonly used protocols to provide secure communications. The protocol allows server/client applications to communicate over a channel that is designed to prevent eavesdropping, tampering, and message forgery. This protocol first appeared under the name *Secure Sockets Layer* (SSL). In 1994 Netscape Communications had developed SSL 1.0, that was never publicly released. In 1995, SSL 2.0 was released, becoming the first release. SSL 3.0 was released in 1996, bringing improvements to its predecessor such as allowing perfect forward secrecy using the Diffie-Hellman key exchange algorithm. The first version of TLS, TLS 1.0, was released in 1999 introducing support for extensions in Client

and Server Hello messages TLS 1.1 and TLS 1.2 were released, respectively in 2006 and 2008, bringing improvements such as reducing CBC block chaining attacks and supporting more block encryption modes for use with Advanced Encryption Standard (AES). In March 2018, TLS 1.3 was approved by the Internet Engineering Task Force (IETF), becoming the new standard for secure connections [4].

However, protocols that allow secure communications may contain vulnerabilities that make them insecure. Over the years, many vulnerabilities have been discovered and corrected in SSL/TLS. The vulnerabilities with which we are concerned can be divided into three groups: design vulnerabilities, implementation vulnerabilities and in cryptographic mechanisms vulnerabilities. The process of updating the software is advisable to fix the vulnerabilities, but sometimes this is not done, e.g., the update process may be time-consuming. The plan of this work consisted of exploring the diversity in cryptographic mechanisms by using multiple cipher suites that allows defining a key exchange algorithm, an authentication mechanism, an encryption mechanism, and a Message Authentication Code (MAC) algorithm. To implement this idea, we intended to used existing protocols and tools without modifying them, because implementing a solution from scratch would be inadvisable, as it could lead to the creation of vulnerabilities, in addition, the existing tools have the advantage of being widely debugged. The reason we do not want to modify existing tools is that we want our solution to be able to use new versions of the tools since these new versions usually have security fixes. Thus, it becomes easy to perform the maintenance to our solution. Taking into account the existing problems and the objectives defined, the solution found consisted in creating several TLS channels, each using a different cipher suite of the other TLS channels, and using tunneling mechanisms to encapsulate each TLS channel within another.

We present MultiTLS, a middleware that uses diversity and tunneling mechanisms through the use of socat version 1.7.3.2 and OpenSSL version 1.1.0g tools to create multiple TLS channels and encapsulates each in other. MultiTLS is a script in bash language and can be run as a shell command. MultiTLS is configured with a parameter k, the *diversity factor* ($k > 1$). This parameter indicates the amount of TLS channels to be created and consequently the number of cipher suites to be used. The cipher suites used by these TLS channels are different from each other, which allows obtaining diversity in the cryptographic mechanisms. In this way, it is possible to mitigate the vulnerabilities that can be found in the cryptographic mechanisms used by TLS channels, including zero-day vulnerabilities which can not be removed as they are unknown [5]. Therefore, the communication channel created by MultiTLS has multiple layers of protection, so that if $k - 1$ of the used cipher suites are considered vulnerable,

communications will remain secure, since there is at least one cipher suite that guarantees the reliability of communications. MultiTLS aims to make progress over vtTLS [6, 7]. vtTLS is a vulnerability-tolerant communication protocol based on diversity and redundancy of cryptographic mechanisms to provide a secure communication channel. vtTLS negotiates more than one cipher suite between server and client, i.e., the communications channels are characterized by not relying on individual cryptographic mechanisms, so that if one is found vulnerable the channels remain secure. The problem of vtTLS is that it modifies an implementation internally, leading to software maintenance challenges, while MultiTLS is always able to use the latest versions - with the latest security fixes. Another difference is that vtTLS uses two cipher suites for the same TLS channel, while MultiTLS encapsulates one TLS channel in another, and the cipher suites used by each TLS channel is different from the others.

1.1 Contributions

The main contribution of this thesis is MultiTLS, a new middleware for secure communication channels that uses diversity and tunneling to tolerate vulnerabilities in cryptographic suites. We present an experimental evaluation of the middleware.

1.2 Outline

The remainder of this document is structured as follows. Chapter 2 presents background and related work. Chapter 3 presents the MultiTLS and its implementation. Chapter 4 presents the evaluation. Finally, Chapter 5 presents the conclusions and future work.

Chapter 2

Background and Related Work

This chapter describes the protocol TLS and its basic information, provides some vulnerabilities in TLS protocol and in cryptographic mechanisms used by it, states related work on approaches to achieve security through diversity as well as presents tunneling mechanisms.

Section 2.1 describes the TLS protocol and the cryptographic mechanisms used by it. Section 2.2 presents vulnerabilities in the SSL/TLS protocol and cryptographic mechanisms. Section 2.3 refers to the advantages of using diversity in security; presents vtTLS, a protocol that provides vulnerability-tolerant secure communication channels; and presents encapsulation mechanisms, as it is the technique to allow the composition of multiple protection layers. Section 2.4 presents the summary of this chapter.

2.1 SSL/TLS

This section describes the SSL/TLS protocol, one of the most commonly used protocols for secure communication channels. The study and understanding of SSL/TLS are very important since MultiTLS uses this protocol. We begin this section by explaining the layers that compose the SSL/TLS protocol: the TLS Record protocol, the TLS Handshake protocol. Ultimately, we present the different configuration variants of the TLS protocol.

2.1.1 The SSL/TLS protocol

The Secure Sockets Layer (SSL) [8] is a security protocol that provides secure communication channels between two entities - server and client. SSL was developed by Netscape Communications in 1994. In order to obtain secure communication, the entities must have communication channels that guarantee authenticity, confidentiality, and integrity. Over the years, SSL was improved and in 1999 the first version of TLS [9], TLS 1.0, the successor to SSL appeared.

TLS 1.2 [10] was the most recent version in the last ten years since in March 2018 TLS 1.3 was approved by the Internet Engineering Task Force (IETF), becoming the new standard for secure connections [4].

As we can see in Figure 2.1, the TLS protocol is structured into two layers: the TLS Record protocol and the TLS Handshake protocol.

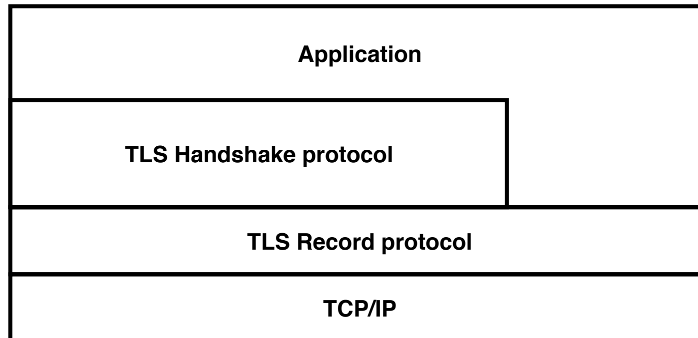

Figure 2.1: TLS protocol stack

TLS Record protocol

The TLS Record protocol is used by the TLS Handshake protocols and the application data protocol to provide mechanisms for sending and receiving messages.

In regard to sending messages, as can be seen in Figure 2.2, the TLS Record protocol starts by fragmenting the message into blocks called TLSPlaintext. These blocks contain four parameters: type, protocol version, the fragment of the layer above and the length of this fragment which must be 2^{14} bytes or less. After the fragmentation step, each TLSPlaintext may be optionally

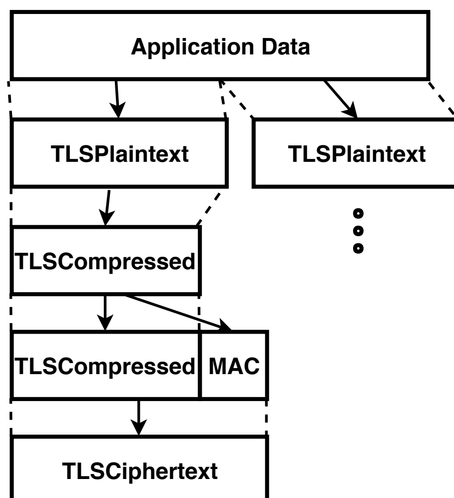

Figure 2.2: TLS Record protocol procedure

compressed into a new block called TLSCompressed. Similar to TLSPlaintext, each TLSCom-

pressed block contains four parameters: type, protocol version (this two as TLSPlaintext), the compressed fragment and the length of this compressed fragment which must be $2^{14} + 1024$ bytes or less. Following the compression step, each TLSCompressed block is processed into a TLSCiphertext block by message authentication code (MAC) and encryption mechanisms. Each TLSCiphertext block contains the type, the version, the encrypted fragment that is encrypted form of the compressed fragment from the previous step, with the MAC, and the length of this encrypted fragment which must be $2^{14} + 2048$ bytes or less. After all these steps, the message can be sent to the destination.

For receiving messages, the process is the inverse of the process described above, that is, decrypted, verified, decompressed, reassembled, and then delivered to the above layer.

Initially, during the first execution of TLS Handshake protocol, the TLS Record protocol does not compress, encrypt, and does not use the MAC, since the server and client have not yet agreed on the algorithms to be used for these actions.

TLS Handshake Protocol

The TLS Handshake protocol is used to establish or resume a secure session between server and client.

To do this, as we can see in Figure 2.3,the TLS Handshake protocol has the following steps:

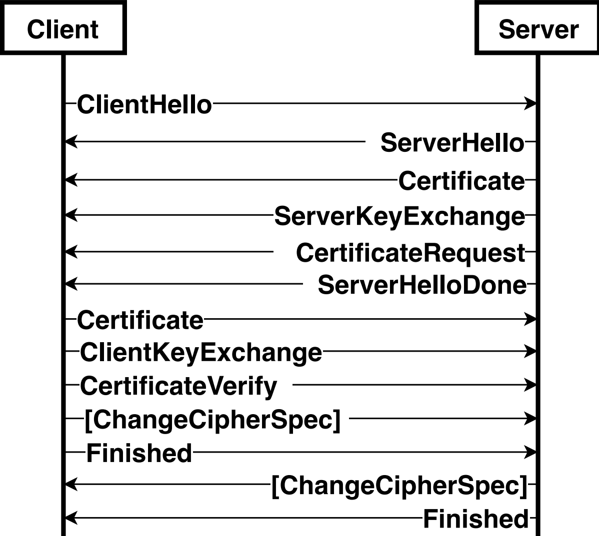

Figure 2.3: Sequence diagram of the TLS Handshake protocol, adapted from [10].

1. The client sends a ClientHello message to the server. This message contains the client's TLS version, a structure called Random (which has a secure random number with 28 bytes and the current date and time), the session identifier, the cipher suite list that contains the combinations of cryptographic mechanisms supported by the client and a list

of compression methods supported by the client, both lists ordered by preference. The client may request additional functionalities by sending a list of extensions. Each cipher suite defines a key exchange algorithm, a bulk encryption algorithm and a hash function.

2. The server responds with a ServerHello message. This message contains the server's TLS version, a structure called Random similar to ClientHello's structure with the same name, the session identifier, one cipher suite and one compression method, both of them chosen by the server from the two lists received in the ClientHello message, and a list of extensions.

3. The server sends Certificate message with its certificate to the client.

4. A ServerKeyExchange message is sent to the client only if the server's certificate does not contain enough information in order to allow the client to share a premaster secret.

5. The server sends a CertificateRequest message to request for the client's certificate. This message is composed by a list of the certificates the client may send, a list of the server's supported signature algorithms and a list of certificate authorities accepted by the server.

6. The server sends a ServerHelloDone message when it concludes its first sequence of messages.

7. The client sends its certificate to the server if the server has sent the CertificateRequest.

8. The client sends to the server a ClientKeyExchange. The content of this message depends on the key exchange mechanism chosen by the server. Some possible content of this message includes the client Diffie-Hellman public value or an encrypted premaster secret.

9. The client sends a CertificateVerify message to the server if the client's certificate has signing capability. The goal of this message is to explicitly verify the client's certificate.

10. The client must send a ChangeCipherSpec message to the server. This message is used to inform the server that the client is now using the agreed upon encryption algorithms. TLS has a specific protocol to indicate changes to cipher suites, designated Change Cipher Spec. This protocol consists of sending a ChangeCipherSpec message by the server and the client to notify each other that they will use the agreed cryptographic algorithms.

11. The client sends Finished message to verify the success of the key exchange and the authentication. This is the first encrypted message sent by the client to the server.

12. After receiving the ChangeCipherSpec message, the server starts using the algorithms established previously and sends a ChangeCipherSpec message to the client. For the reason that receiving this message causes the receiver to instruct the TLS Record protocol to use the newly negotiated cryptographic algorithms and keys in the received messages.

13. The server terminates the handshake protocol by sending its first encrypted message, Finished ,message to the client.

Following the TLS Handshake protocol, the session is established. The server and the client can exchange information through the secure communication channel.

2.1.2 TLS configuration variants

As previously mentioned, TLS allows the server and client to establish secure communications channels, giving them the freedom to choose the cryptographic algorithms to be used. To do this, the communication entities have to choose one cipher suite.

Each cipher suite defines:

- one key exchange algorithm - specifies an asymmetric cryptography algorithm that is used to generate shared secrets;

- one bulk encryption algorithm - defines the symmetric key algorithm, the secret key length, and the cipher mode are used to encrypt and decrypt the messages exchanged between the server and the client;

- one hash function - indicates the message authentication code (MAC) algorithm to verify the data integrity and the authentication of a message.

For example, the cipher suite TLS_RSA_WITH_AES_128_GCM_SHA256 is indicated that the key exchange algorithm is RSA, the bulk encryption algorithm is Advanced Encryption Standard (AES) with 128-bit key length and using Galois/Counter Mode (GCM), and the hash function is the SHA256. Table 2.1 presents the key exchange algorithms supported by TLS. Table 2.2 presents the symmetric key algorithms supported by TLS. Table 2.3 presents the hash functions supported by TLS.

A keyed-hash message authentication code (HMAC) is an algorithm that combines a secret key with MAC algorithm to verify data integrity and authenticity of a message. There is also a mechanism called authenticated encryption with associated data (AEAD) [11] that simultaneously provide confidentiality, integrity, and authenticity.

Algorithm	Description	Perfect Forward Secrecy (PFS)	Secure
RSA	RSA key exchange algorithm	✗	✓
DH-RSA	Diffie-Hellman (DH) with RSA-based certificates	✗	✓
DH-DDS	DH with DSS-based certificates	✗	✓
DHE-DSS	Ephemeral DH with DDS (DSA) signatures	✓	✓
DHE-RSA	Ephemeral DH with RSA signatures	✓	✓
DH-anon	Anonymous DH without signatures	✗	✗
ECDH-RSA	Elliptic curve (EC)DH with RSA signatures	✗	✓
ECDH-ECDSA	ECDH with ECDSA (variant of DSA)	✗	✓
ECDHE-RSA	Ephemeral ECDH with RSA signatures	✓	✓
ECDHE-ECDSA	Ephemeral ECDH with ECDSA (variant of DSA)	✓	✓
RSA-PSK	Pre-shared key exchange with RSA	✗	✓
DHE-PSK	Pre-shared key exchange with ephemeral DH	✓	✓
ECDHE-PSK	Pre-shared key exchange with ECDHE	✓	✓
ECDHE-anon	Anonymous ECDHE without signatures	✗	✗

Table 2.1: Key exchange algorithms supported by TLS 1.2. If the algorithm provides forward secrecy, it is marked with a (✓) in the third column. In case the algorithm is regarded as insecure, it is marked with a (✗) in the Secure column.

Cipher	Mode opetation	**Type**	**Key size**	**Secure**
AES	Galois/Counter Mode (GCM)	Block	128, 256	✓
	Counter with CBC-MAC (CCM)			✓
	Cipher Block Chaining (CBC)			✓
3DES EDE	CBC	Block	112	✗
Camellia	GCM	Block	128, 256	✓
	CBC			✓
ARIA	GCM	Block	128, 256	✓
	CBC			✓
SEED	CBC	Block	128	✓
ChaCha20	———	Stream	256	✓
RC4	———	Stream	128	✗

Table 2.2: Symmetric-key cipher algorithms supported by TLS 1.2. In case the algorithm is regarded as insecure, it is marked with a (✗) in the Secure column.

Hash function	Digest size	Block size	Rounds	Collisions found
MD5	128	512	64	✗
SHA (SHA-1)	160	512	80	✗
SHA-256 (SHA-2)	256	512	64	✓
SHA-384 (SHA-2)	384	1024	80	✓

Table 2.3: Hash functions supported by TLS 1.2 to verify data integrity and authenticity.

2.2 TLS Vulnerabilities

Although the TLS protocol aims to establish secure communication channels, it may contain vulnerabilities making these channels insecure and susceptible to attacks. To prove the importance of our work in increasing communications security, we will present some vulnerabilities found in the TLS protocol and in some cryptographic algorithms used by it.

According to the Internet Security Glossary, Version 2 [12], vulnerabilities can be classified into three groups: design vulnerabilities, implementation vulnerabilities, and operation and management vulnerabilities. We are focused only on these first two groups of vulnerabilities. The design vulnerabilities refer to protocol specification failures. Releasing a new version or update is the only way to fix this type of vulnerability. The implementation vulnerabilities are related to failures that were created during the implementation phase of the protocol, that is, it is in the code of a TLS implementation.

In the following sections, some design and implementation vulnerabilities will be exposed. In addition to these vulnerabilities, we will also present vulnerabilities in cryptographic algorithms since they play an important role in the TLS protocol and also because our solution will use combinations of different cryptographic mechanisms to achieve diversity and thereby increase security in the communication channels.

2.2.1 Design vulnerabilities

An example attack that exploits a design vulnerability is Compression Ratio Info-leak Made Easy (CRIME). This vulnerability was found in TLS compression. The main purpose of compression is to reduce the size of messages to be transmitted, while preserving their integrity. DEFLATE is the most common compression algorithm used. One of the techniques used by compression algorithms is to replace repeated bytes with a pointer to the first instance of that byte. If a victim and server are using the DEFLATE compression method and if an attacker knows that for the session the targeted website creates a cookie called "user" then the attacker can obtain the victim's cookie through a man-in-the-middle attack (MITM), so the attacker needs to inject "Cookie: user = 0" into the victim's cookie, the server will only append the character "0" to the compressed response since "Cookie: user =" is already sent in the victim's cookie. All the attacker must do is inject different characters and then monitor the size of the response. If the response size is smaller than the initial one, it means that the character they injected is contained in the value of the cookie and thus has been compressed, which is equivalent to a match. If the character is not in the cookie value, the response size will be larger. Using this method, an attacker can brute-force the cookie value by using the responses sent by the server.

In August 2016, was presented, one of the most recent attacks against a specification vulnerability, the DROWN (Decrypting RSA with Obsolete and Weakened eNcryption) attack [13]. This attack exploits weaknesses of SSL 2.0. Although there are newer and more secure TLS versions, due to incorrect configurations and compatibility, many servers still support older SSL/TLS versions. By itself, this is not a risk, since no updated customer actually uses SSL 2.0. DROWN proves that simple SSL 2.0 support from servers is a threat to TLS communication channels that use the recent versions of TLS. The attack allows an attacker to decrypt the messages exchanged between servers and clients updated by sending probes to a server that supports SSL 2.0 and uses the same private key. In several steps, and using different techniques, such as Bleichenbacher iteration technique [14], the attacker can decrypt a message.

2.2.2 Implementation vulnerabilities

In 2014, an implementation vulnerability was discovered in OpenSSL [15], called Heartbleed. The name of the vulnerability is related to an extension where a vulnerability appears, the heartbeat extension [16], which is an extension to the TLS protocol designed to enable a low-cost, keep-alive mechanism for client and server to know that they are still connected. The extension consists of sending a message with an arbitrary payload and the size of that same payload. After the receiver receives this message, it returns the received payload. The Heartbleed vulnerability [17] is a buffer over-read vulnerability that happens when the sender sends a message that specifies a payload size higher than what payload really has. The receiver upon receiving the message returns a block of memory where the sent payload begins plus the specified size of the received message, that is, it returns the received payload plus a dataset with size equal of the size specified in the received message minus the real size of the message. This allows an attacker to obtain a dataset that may contain, for example, authentication secrets, such as session cookies and passwords.

2.2.3 Cryptographic vulnerabilities

This section presents some vulnerabilities in cryptographic mechanisms, specifically in some of the mechanisms supported by the TLS protocol. Not every mechanism supported by TLS protocol is secure, as it may be in Section 2.1.2, but remains in the protocol to maintain compatibility.

Our solution use diverse cipher suites as a form to increase security. For this, it is necessary to study the vulnerabilities in the cryptographic mechanisms in order to know which cipher suites are more secure and which could be used.

Vulnerabilities in asymmetric cipher mechanisms

RSA [18] proposed by Rivest *et al.*, in 1978, is an asymmetric cryptographic algorithm used to cipher and sign messages. RSA's security is based on two problems: integer factorization problem and the RSA problem [19]. The integer factorization problem consists of the decomposition of a number into a product of smaller integers that must be prime numbers. RSA with key size equal to 768 bits (RSA-768) is unsafe after Kleinjung et al. have factored a number with 768 bits, equivalent to a number with 232 digits [20]. Although the use of RSA-1024 is currently discouraged, no factorization has yet been published.

Shor's algorithm [21] factorizes integers in polynomial time, making the integer factorization problem easy to solve. However, this algorithm requires for quantum computers, something that does not yet exist in practice.

Another possible attack on RSA is the Chosen Plaintext Attack (CPA) [22]. This attack explores the fact that RSA is deterministic, that is, a specific plaintext always originates the same ciphertext. A chosen plaintext attack consists of an attacker sending an unlimited number of plaintext messages of his own and examining the resulting cryptograms so that he compares them with a captured ciphertext and discovers the original message.

A Chosen Ciphertext Attack (CCA) [23] is an attack that can also be used against RSA, where an attacker can collect information by obtaining the decryptions of chosen ciphertext messages. With this information, the attacker can attempt to recover the hidden secret key used for decryption.

Vulnerabilities in symmetric cipher mechanisms

Triple DES (3DES) is a symmetric key block cipher which consists of executing three times the Data Encryption Standard (DES) [24]. 3DES can use the same key for the three rounds or use two different keys, i.e. one of them will be used in two rounds or use different keys for each round. All three options have vulnerabilities.

The first option has the same security as DES, which makes it an insecure option. This is due to the fact that the size of the key is too small.

Keying option two uses two different keys with 56 bits, that is, a key with 112 bits. This option can be attacked using CPA with 2^{56} operations [25].

The option that uses the three different 56-bit keys, i.e., a key with 168 bits, is vulnerable to meet-in-the-middle attack which requires 2^{112} steps to attack 3DES. This attack can be reduced to 2^{108} [26]. This means that 3DES is just twice as secure as DES and not three times. 3DES is usually defined by $E^1 = E^3 = E$, $E^2 = D$, where E denotes the DES encryption function,

D denotes the decryption function and the exponent represents the round. Let a plaintext and ciphertext pair (p, c) be given, the attack proceed as follows:

1. Compute all values $b_N = D_N^3(c), N \in \{0,1\}^k$, and store the pairs (b_N, N) in a table;
2. Compute all values $b_{L,M} = E_M^2(E_L^1(p))$ with L, M $\in \{0,1\}^k$, and search for (b_N, N) where $b_{L,M} = b_N$;
3. Test all key triples (L,M,N) with $b_{L,M} = b_N$ until only one such triple remains.

The Advanced Encryption Standard (AES) is an encryption algorithm created by Rijmen and Daemen [27]. The key used in AES can have one of three different sizes - 128, 192, or 256 bits. The size of the key influences the number of rounds that are, respectively, 10, 12 and 14. In 2011, Bogdanov *et al.* [28] published biclique attack against AES, though only with slight advantage over brute force. The computational complexity of the attack is $2^{126.1}$, $2^{189.7}$ and $2^{254.4}$ for AES128, AES192 and AES256, respectively. Although, there is this attack and others, AES is still considered a secure encryption mechanism.

Vulnerabilities in hash functions

A hash function, sometimes also called message digest function, is an algorithm that transforms variable length data into smaller data sets with a fixed length called hash values, checksums or simply hashes. A hash function is required to satisfy the following properties [19]:

- Easy to compute the hash value for any given message;
- Preimage resistance - infeasible to generate a message that has a given hash value;
- Second preimage resistance - infeasible to modify a message without changing the hash value;
- Collision resistance - infeasible to find two different messages with the same hash.

Thus, the hash functions can be interpreted as a special compression of the message that works like a fingerprint of the message, making its use useful for data integrity and message authentication. Note that it is impossible to have a unique identity once the message is compressed, allowing attackers break the collision resistance property.

MD5 [29] is a hash function, created by Rivest in 1991, that produces a 128 bit hash. In 2005, MD5 was proved not collision resistant by X. Wang and H. Yu [30]. They proved it through differential attacks, more specifically a modular differential attack. The differential

cryptanalysis, introduced by E. Biham and A. Shamir [31], is a method which analyzes the effect of differences in input pairs on the differences of the resultant output pairs.

The Secure Hash Algorithm 1 (SHA-1) is another hash function, published by the United States NIST in 1995, which produces a 160 bit hash. In 2017, M. Stevens et al. demonstrated that the theoretical attacks on SHA-1 are practical when displaying the first example of a collision for SHA-1 [32]. The computational effort spent on the attack is estimated to be equivalent to $2^{63.1}$ SHA-1 calls.

Nowadays, SHA-2 and SHA-3 are the recommended hash functions to use instead of of the SHA-1 [33].

2.3 Achieving security through diversity

A static system is characterized by no changes over time and therefore an attacker has time to discover vulnerabilities in the system. In order to overcome the problems caused by static defense mechanisms, moving target defense was proposed as a way to make it more difficult for an attacker to exploit vulnerabilities of a system, through dynamic defense mechanisms [34]. Moving target defenses can classify into two groups: proactive and reactive [35]. Proactive moving target defenses adapt to a specific schedule, without feedback from the system. Reactive moving target defenses make changes in the protected system when they receive a notification from a security sensor.

The term *diversity* describes multi-version software in which redundant versions are purposely made different from between themselves [36]. Multiple copies of a program contain the same faults, causing low protection. With diverse versions, one hopes that any faults they contain will be different and show different failure behavior.

A specification of a program can be implemented in various different forms since each programmer has his way of thinking and implementing the program according to the specification. When diversity must be introduced and what must be diversified, it must be the questions that the programmer must ask himself when he wants to diversify his software. The diversity does not change the logic of the program.

By reason of our solution uses encapsulation mechanisms to achieve diversity and, therefore, security, studying the various encapsulation protocols helps us to understand better these techniques and how to use them.

2.3.1 Automated software diversity

Automated software diversity are techniques that make the exploitation and execution of vulnerabilities more difficult. The objectives of software diversity are to make unpredictable the features of the program and hide them from opponents.

Transparent Runtime Randomization (TRR) [37] is an approach for protecting against many security attacks, such as buffer overflow and format strings. TRR dynamically and randomly relocates a program's stack, heap, shared libraries, and parts of its runtime control data structures inside the application memory address space. Once, the memory layout of the program is different each time that program is run, the attacker will have difficulty in determining the values of critical addresses.

Replicas have an important role in a reliable distributed system. However, replicas normally use the same code causing them to share the same vulnerabilities and, therefore, do not exhibit independence to attack. Proactive obfuscation [38] is a mechanism that restores some measure of this independence by restarting each replica periodically with a newly generated and diverse executable. Proactive obfuscation has three components, as can be seen in the Figure 2.4.

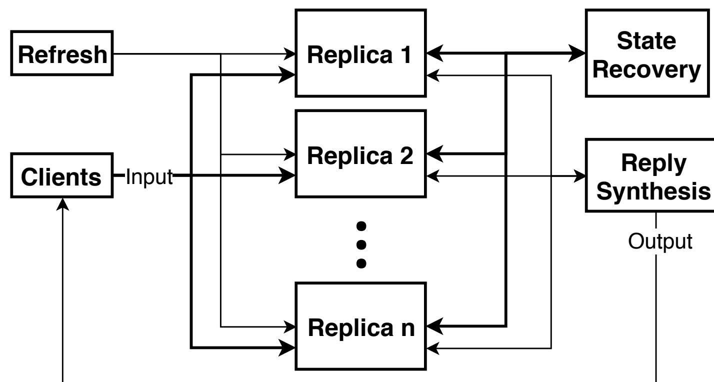

Figure 2.4: Representation of a replicated service with n replicas and the proactive obfuscation.

Reply Synthesis is used to transform the output from many replicas into an output from the replicated service in consideration of the minimum number of distinct replicas in the system. The State Recovery allows to recover the state after the replicas are rebooted. The Refresh provides freshly executables to replicas and periodically reboots them.

Attacks that reuse code are the most difficult to defend. The solution to this problem can focus on the use of automated diversity of software. Profile-oriented optimization focuses on that part of the program code, where a program spends most of its runtime. The use of profile-oriented optimization reduces the performance overhead of software diversity. The use of NOP insertions [39] is an approach that applies these concepts as a way of solving problems related to reused code and code injection.

2.3.2 Vulnerability-Tolerant TLS

Another alternative to achieve diversity is to use vtTLS. This is a protocol that provides vulnerability-tolerant communication channels. The protocol aims to solve the problem of TLS, originated by having only one cipher suite negotiated between server and client. In these cases, if one of the cryptographic mechanism of cipher suite becomes insecure, the communication channels using this cipher suite may become vulnerable The idea was to use the diversity and redundancy of cryptographic mechanisms, keys and certificates. The communication channels created by vtTLS are characterized by establishment of k cipher suites, so that if vulnerabilities are found in the $k-1$ cipher suites cryptographic algorithms, the channels will still remain secure due to the remaining cipher suite. vtTLS was implemented as a modification of OpenSSL version 1.0.2g, which causes some maintenance problems, since moving to another version of OpenSSL requires implementing the diversity features again but in the new version of OpenSSL which is quite a challenge. However, the VTTLS and TLS handshake protocols are similar. The message names are identical to facilitate the transition from TLS. The messages that require the use of diversity are ServerHello, ServerKeyExchange, (Server and Client) Certificate, ClientKeyExchange.

In the ServerHello message, the server sends to the client the k cipher suites to be used in the communication. These cipher suites were chosen from the list previously sent by the client in the ClientHello message.

In the ServerCertificate message, the server sends all its k certificates to the client. Each certificate is associated with a key exchange mechanism (KEM). Consequently, the k cipher suites must use the key exchange mechanisms supported by the certificates of the server.

The ServerKeyExchange message is sent only if one of the k cipher suites includes a key exchange mechanism that uses ephemeral keys. The contents of this message are the server's DH ephemeral parameters. For each cipher suite that uses DH ephemeral parameters, the server sends a ServerKeyExchange message with different DH ephemeric parameters.

In the ClientCertificate message, all certificates of the client are sent to the server. Next sending its certificates, the client sends k ClientKeyExchange messages to the server. The content of these messages is based on the k cipher suites agreed. If m of the cipher suites use RSA as KEM, the client will send m messages, each with an RSA-encrypted pre-master secret to the server ($0 \leq m \leq k$). If j of the cipher suites use DH ephemeric parameters, the client sends j messages to the server containing these parameters ($0 \leq j \leq k$). If some of the k cipher suites have the same KEM, the use of different parameters for each cipher suite allows for greater diversity and thus more security.

Our solution is similar to this approach but we do not modify implementations of the tools. This form our solution is always able to use the latest versions - with the latest security fixes.

2.3.3 Tunneling

The term *tunneling* describes a process of encapsulating entire data packets as the payload within others packets, which are handled properly by the network on both endpoints [40]. The tunneling process requires three different types protocols with the following function:

- Carrier protocol - the network protocol used to transport the final encapsulation;
- Encapsulating protocol - the protocol used to provide the new packet around the original data packet;
- Passenger protocol - the original data packet that is been encapsulated.

This section introduces some tunneling protocols and analyzes its functioning with regard to security.

Point-to-Point Protocol (PPP) is a data link protocol that provides a standard method for transporting multi-protocol datagrams over point-to-point links [41].

The PPP encapsulation provides for multiplexing of different network-layer protocols simultaneously over the same link. PPP encapsulation requires the following three fields: Protocol, Information, and Padding. The Protocol field identifies the encapsulated protocol in the Information field. The Information field may be padded with an arbitrary number of bits up to the Maximum Receive Unit (MRU).

In PPP the peers' authentication is not mandatory. However, if needed the peers can be authenticated by using Password Authentication Protocol (PAP) or Challenge Handshake Authentication Protocol (CHAP) [42]. PAP is a password-based authentication protocol considered weak due to the fact that it transmits unencrypted messages over the network. On other side, CHAP [43] is an authentication protocol more secure composed by three steps:

1. The authenticator sends a challenge to the peer;
2. The peer sends a value which is calculated using a hash function on the challenge and the secret combined;
3. The authenticator verifies if the value received is the expected hash value calculated by authenticator. If the values match, the authenticator recognizes the authentication, otherwise it should terminate the connection.

Another option for authentication over PPP is Extensible Authentication Protocol (EAP). EAP is an authentication framework, which supports multiple authentication methods, was created as an alternative to PAP and CHAP. Later they were incorporated in EAP.

Beyond authentication configurations, PPP also supports transmission encryption using the Encryption Control Protocol (ECP) [44]. An encrypted packet is encapsulated in the Information field, where the Protocol field indicates hexadecimal value 0053 to indicate encrypted datagram. PPP DES Encryption Protocol (DESE) [45], PPP Triple-DES Encryption Protocol [46] and DESE-bis [47] are the algorithms used to encrypt.

The Point-to-Point Tunneling Protocol (PPTP) [48] is a protocol that allows PPP connections to be tunneled through an IP network, creating a Virtual Private Network (VPN). VPNs enable a secure connection within a public network, i.e., a VPN extends a private network over a public network and it allows users to send and receive messages over public networks as if their computing devices were directly connected to the private network.

The virtual network packets is encapsulated in PPP packets, which are in turn encapsulated in Generic Routing Encapsulation (GRE) [49] packets packets and sent over IP to the other endpoint. PPTP does not define algorithms for authentication and encryption, instead, it provides a framework for negotiating specific algorithms. This negotiation relies upon existing PPP option negotiations [50].

Layer Two Tunneling Protocol (L2TP) [51] is originated primarily by Layer 2 Forwarding Protocol (L2F) and PPTP. L2F is a tunneling protocol developed by Cisco Systems, which is similar to PPTP. L2TP does not encrypt data nor authenticate the messages. However, by combining L2TP with Internet Protocol Security Protocol (IPsec) it is possible to provide an additional layer of authentication and encryption, since L2TP packets are encapsulated in IPSec packets at the network.

The Internet Protocol (IP) transmits block of data called datagrams from sources to destinations, which are hosts identified by addresses [52].

In the IP header of the packets there is a field, called Protocol, to identify the next level protocol [53]. In this field we can used the IP in IP Tunneling protocol.

In IP in IP Tunneling [54], the original header is preserved, and simply wrapped in another standard IP header. An outer IP header is added before the original IP header. Between them are any other headers for the path, such as security headers specific to the tunnel configuration. The outer IP header source and destination identify the endpoints of the tunnel. The inner IP header source and destination identify the original sender and recipient of the datagram (see Figure 2.5).

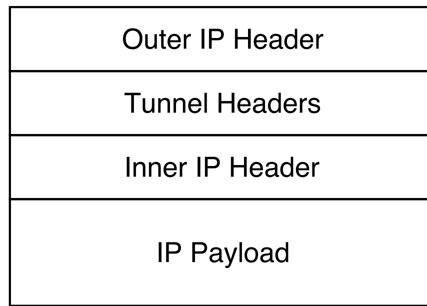

Figure 2.5: The format of the IP in IP encapsulation, adapted from [54].

IPsec [55] is a network protocol suite that authenticates and encrypts the packets sent over a network. IPsec has two encryption modes: tunnel and transport. Tunnel mode encrypts the header and the payload of each packet while transport mode encrypts the payload.

IPsec uses the following protocols to perform various functions:

- Authentication Headers (AH) provide authentication and data integrity for IP datagrams;
- Encapsulating Security Payloads (ESP) provide confidentiality, authentication and message integrity.

Figure 2.6 represents the packets encapsulated with AH and ESP in the two modes and show which parts are authenticated e ciphered.

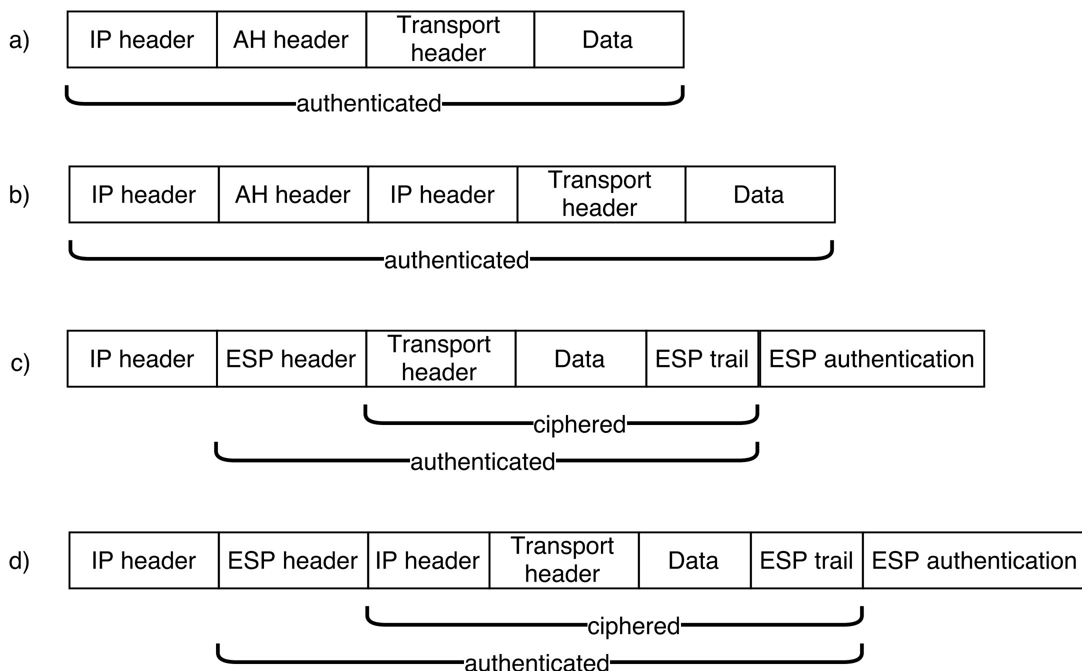

Figure 2.6: Structure of the use of IPsec mechanisms. a) IPsec AH in Transport mode. b) IPsec AH in Tunnel mode. c) IPsec ESP in Transport mode. d) IPsec ESP in Tunnel mode

The Secure Shell Protocol (SSH) is a protocol for secure remote login and other secure

network services over an insecure network [56]. SSH is typically used to log into a remote machine and execute commands, but it also supports tunneling.

SSH tunneling provides a secure channel. Figure 2.7 show an application that wants to contact the server, it will connect with a SSH client who will handle the communication to the SSH server. Then the SSH server, that is on the same secure network to the server, sends the message to the server.

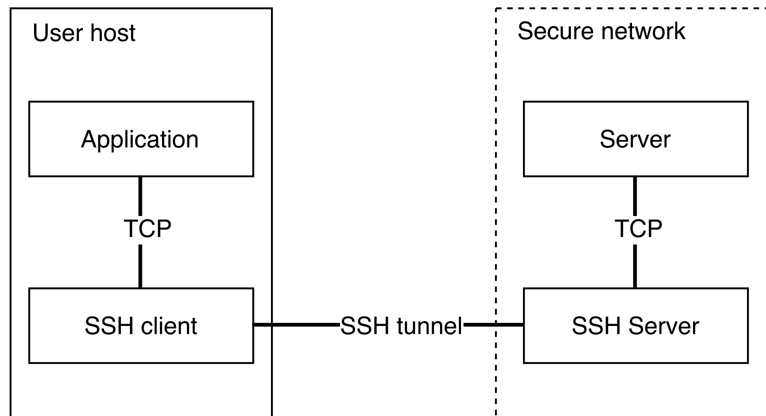

Figure 2.7: Representation of the SSH tunnel configuration

SSH may have been divided into the following three layers:

- Transport layer protocol - it provides encryption, server authentication, and integrity protection [57];
- Authentication protocol - it runs on top of the Transport layer protocol and provides mechanisms that can be used to authenticate the client to the server [58];
- Connection protocol - it also runs on top of the Transport layer protocol and specifies a mechanism to multiplex multiple channels over the confidentiality and authentication transport [59].

These layers provide mechanisms that make SSH secure for tunneling.

Although there are several protocols that encapsulate other protocols, there is no known case of SSL/TLS encapsulation, i.e., of tunneling TLS over TLS.

2.4 Summary

In this chapter, we presented the TLS protocol and its two sub-protocols, the TLS handshake protocol and the TLS Record protocol. We presented the concept of cipher suite and which cryptographic algorithms can be specified in a cipher suite. It addressed the vulnerabilities in

the design and the implementation of the TLS protocol, as well as the cryptographic mechanisms used by TLS. It was presented the advantages of diversity and some techniques that use it as a way to obtain security. It was approached the vtTLS, a protocol that allows to create channels of communication tolerant to vulnerabilities. Finally, we have introduced some tunneling mechanisms since our solution uses tunneling mechanism to encapsulate several TLS channels, each in another.

Chapter 3

MultiTLS

MultiTLS is a middleware that provides secure communication channels with multiple layers through the creation and tunneling of TLS channels within each other. It provides an increase in security since each of these TLS channels uses a different cipher suite than the others. As mentioned before, TLS channels individually use only one cipher suite, which consists of a single point of failure if the cryptographic mechanisms used become vulnerable. MultiTLS solves this problem by allowing the server and the client to create a communication channel composed by k TLS channels, with $k > 1$, and consequently also allows to use k cipher suites and certificates, in contrast to a communication that uses only one TLS channel.

The reason MultiTLS achieves increased security is that even when $(k-1)$ cipher suites become insecure, that is, even when $(k-1)$ TLS channels become vulnerable, the communication channel created by MultiTLS, which is the combination of the k TLS channels, remains secure since there is still one TLS channel with secure cipher suite. The mechanisms used by MultiTLS allow to create k TLS channels and encapsulate one into another without changing the implementations of the tools used. This means that it is easy to maintain since switching to newer versions of the tools, which usually have security fixes, does not interfere with the implementation of MultiTLS. This approach is an advantage over vtTLS, since it changes the OpenSSL implementation.

Section 3.1 presents the TUN interfaces, which are the mechanism used by MultiTLS to encapsulate the various TLS channels. Section 3.2 presents the study that allowed to choose the combination of encryption packages supported by MultiTLS and that guarantee a greater diversity. Section 3.3 explains the MultiTLS commands and its arguments. Section 3.4 presents the implementation of MultiTLS, revealing the tools used by it and how it creates the encapsulated TLS channels in the tunnels. Finally, Section 3.5 presents the summary of this chapter.

3.1 Design

To encapsulate a TLS channel on another TLS channel, we use TUN (network TUNnel) interfaces. This mechanism is a feature offered by some operating systems and unlike the common network interfaces, TUN does not have physical hardware components, that is, it is virtual network interface implemented and managed by the kernel itself. TUN is a virtual point to point network device whose driver was designed as low level kernel support for IP tunneling. It works at the protocol layer of the network stack. TUN interfaces allow user-space applications to interact with them as if they were a real device, remaining invisible to the user. These applications pass packets to a TUN device, in this case, the TUN interface delivers these packets to the operating system's network stack. Conversely, the packets sent by an operating system to a TUN device are delivered to a user-space application that attaches to the device. Figure 3.1 shows a practical example in which an application running on two different hosts communicates through TUN interfaces.

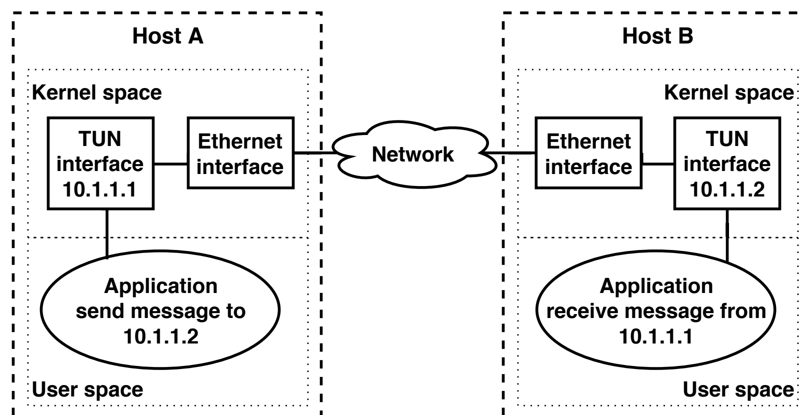

Figure 3.1: Example of using TUN interfaces

A tunnel can be described as the communication channel between the TUN interfaces and that encapsulates the communications that use these interfaces. By creating TUN interfaces through others created previously, we create an encapsulation of several tunnels. For each of these interfaces, we can use TLS implementations running in user space that allows creating a TLS channel that is encapsulated by the tunnel between the interfaces created on different hosts. Figure 3.2 presents the MultiTLS for $k = 2$. This configuration allows an application to communicate over two tunnels, whereas the tunnel between the TUN1 interfaces encapsulates the tunnel between the TUN2 interfaces. In addition, we can see that between the TUN 1 interfaces there is a tunnel that crosses two processes that we designate by TLS implementation and whose function is to establish and manage the TLS channel that is encapsulated by the tunnel. To do this, one of these processes will run in server mode and the other in client mode,

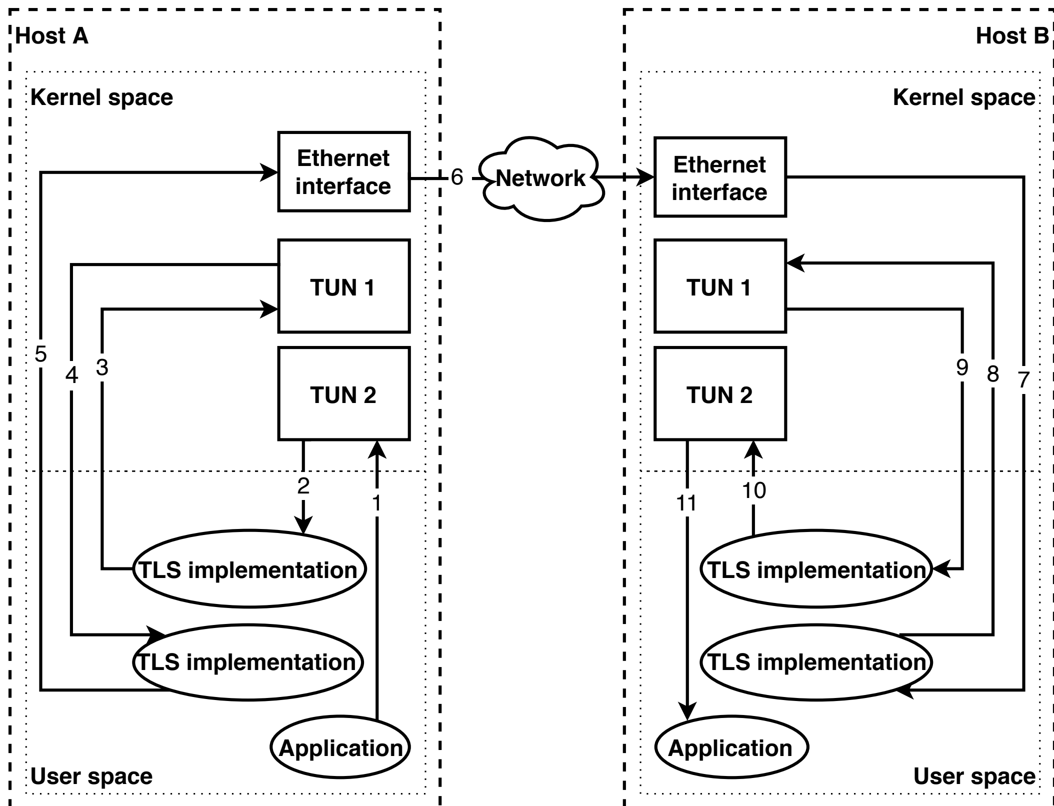

Figure 3.2: MultiTLS design with $k = 2$ and the flow of sending messages from one application to another on different hosts.

as explained in Section 2.1.1.

3.2 Combining Diverse Cipher Suites

We are interested that MultiTLS provides the maximum possible diversity among cryptographic mechanisms, because we want to prevent them from having common vulnerabilities. Evaluating the diversity among cryptographic mechanisms is not trivial. For this purpose, we based on the work of Carvalho [60], regarding the heuristics of comparing diversity among cryptographic mechanisms. In this study, we are focused on searching for the combination of four cipher suites supported by TLS 1.2 from the OpenSSL 1.1.0g implementation, that guarantees greater diversity.

We begin by evaluating the diversity of public key mechanisms. In this case, we observe the various combinations of key exchange and authentication algorithms in cipher suites. The insecure cryptographic mechanisms were discarded as well as the ECDH and DH algorithms since there are the variants of them, ECDHE and DHE, which guarantee perfect secrecy.

This analysis resulted in the following combinations:

- ECDHE for key exchange and ECDSA for authentication;
- RSA for key exchange and authentication;
- DHE for key exchange and DSS for authentication;
- ECDHE for key exchange and RSA for authentication;
- DHE for key exchange and RSA for authentication.

In order to avoid that the key exchange and authentication algorithms are repeated consecutively and since the use of elliptic curve results in a fast computation and low power consumption, we choose the first four combinations of the above list, keeping the presented order, i.e., the first tunnel will use ECDHE for key exchange and ECDSA as authentication algorithm, the second RSA for key exchange and authentication, the third DHE for key exchange and DSS for authentication and the fourth DHE for key exchange and RSA for authentication.

Considering the combination of key exchange and authentication algorithms, we group the supported cipher suites according to this combination, as shown in Table 3.1.

ECDHE_ECDSA	RSA	DHE_DSS	ECDHE_RSA
AES_256_GCM_SHA384	AES_256_GCM_SHA384	AES_256_GCM_SHA384	AES_256_GCM_SHA384
CHACHA20_POLY1305_SHA256	AES_256_CCM_8	AES_128_CBC_SHA256	CHACHA20_POLY1305_SHA256
AES_256_CCM_8	AES_256_CCM	AES_256_CBC_SHA256	AES_128_GCM_SHA256
AES_256_CCM	AES_128_GCM_SHA256	CAMELLIA_256_CBC_SHA256	AES_256_CBC_SHA384
AES_128_GCM_SHA256	AES_128_CCM_8	AES_128_CBC_SHA256	CAMELLIA_256_CBC_SHA384
AES_128_CCM_8	AES_128_CCM	CAMELLIA_128_CBC_SHA256	AES_128_CBC_SHA256
AES_128_CCM	AES_256_CBC_SHA256		CAMELLIA_128_CBC_SHA256
AES_256_CBC_SHA384	CAMELLIA_256_CBC_SHA256		
CAMELLIA_256_CBC_SHA384	AES_128_CBC_SHA256		
AES_128_CBC_SHA256	CAMELLIA_128_CBC_SHA256		
CAMELLIA_128_CBC_SHA256			

Table 3.1: Cipher suites grouped by the combination of key exchange and authentication algorithms

In Table 3.1, to simplify only we put in the title of the columns the expression of the cipher suite that represents the key exchange and the authentication algorithms and underneath we only list the cipher suites expressions referring to the symmetric key algorithms and the hash function. However, the cipher suites expression is obtained by concatenating "TLS_" with the column title concatenated with "_WITH_" plus one of the rows of the respective column, e.g., in the first column, the cipher suite obtained with the first expression listed is TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384.

After this step, we chose in each group the cipher suite that maximizes the diversity of the symmetric key algorithms and the hash function between each of the four groups. In order to measure the diversity of the cryptographic mechanisms, we have taken into account some characteristics such as the origin, i.e., the author or institution that proposed the algorithm, the

Cipher	Type	Mode of operation	Designers	Year	Key size	Rounds	Block size	Structure
AES	Block	GCM	V. Rijmen, and J. Daemen (Belgium)	1998	256	14	128	Substitution permutation networt
					128	10		
		CBC			256	14		
					128	10		
Camellia		CBC	Mitsubishi Electric, and NTT (Japan)	2000	256	24	128	Feistel network
					128	18		
ChaCha20	Stream	—	D. Bernstein (USA)	2008	256	20	—	Add-Rotate-XOR (ARX)

Table 3.2: Diversity metrics of MultiTLS symmetric cryptographic mechanisms

year in which it was designed, the size of the key in the case of the symmetric key algorithms and the digest size in the case of hash functions and other metrics addressed in Carvalho's research.

Table 3.2 presents some information about the symmetric key algorithms. In this way we can conclude that the combinations of 4 symmetric key algorithms that maximize the diversity itself are:

- ChaCha20 + Camellia 256 + AES256-GCM + AES128CBC
- ChaCha20 + Camellia 256 + AES256-CBC + AES128GCM
- ChaCha20 + Camellia 256 + Camellia128 + AES256-GCM

Regarding hash functions, we can see in Table 3.1 that the variety is greatly reduced since there is only SHA-256 and SHA-384. However, some symmetric key algorithms use modes of operation, such as CBC-MAC (CCM mode) [61] and Galois/Counter Mode (GCM), that provide authenticated encryption with associated data (AEAD). It is considered an alternative mechanism which can be used redundantly with HMAC to achieve even higher diversity. In addition, the cipher suites with the ChaCha20 algorithm use the Poly1305 which is a one-time authenticator [62]. Poly1305 takes a 32-byte one-time key and a message and produces a 16-byte message authentication code (MAC).

From these analyses, the cipher suites selected to be used by default in MultiTLS with $k \leq 4$ are:

- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_RSA_WITH_AES_128_CCM_8
- TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

If the MultiTLS user wants to use only 2 tunnels, i.e., k = 2, the first cipher suite shown in the above list is used in the first tunnel and the second cipher suite is used in the second tunnel.

3.3 Running MultiTLS

MultiTLS is a script in bash language and can be run as a shell command. Before presenting how MultiTLS creates the secure tunnels, we will first introduce the commands that allow us to create them and associate the establishment of a TLS channel. The commands available through MultiTLS are:

- `multitls -s port nTunnels [cert cafile]`
- `multitls -s port nTunnels [cert cafile cipher]`
- `multitls -c port nTunnels IPServer [cert cafile]`
- `multitls -c port nTunnels IPServer [cert cafile cipher]`

The flags -s and -c mean that MultiTLS will run as a server or client, respectively. The port argument specifies the port used to establish the last tunnel. In the case of the server MultiTLS will be listening on that port. In the case of the client, MultiTLS will connect to that port of the machine that has the IP specified in the IPServer argument. The nTunnels argument specifies the number of tunnels that MultiTLS will create. In addition, we must specify in the cert argument the path to the file with its certificate and private key, in the cafile argument specifies the file that contains the peer certificate. The cipher argument lets us specify one or more cipher suites. The arguments between brackets must be specified the number of times that the value of the nTunnels argument has.

For better understanding, we will describe an example giving the MultiTLS commands that we need to execute for a specific scenario. We assume that we have two different hosts, one playing the role of a server and the other of a client. Using MultiTLS, we want to establish a communication between the two hosts with two protection layers, that is, $k = 2$. The IP of the server is 192.168.1.119 and will use port 11444 to establish communication with the client. The server has the files containing its certificate and its private keys for the first tunnel, S1.pem, and for the second tunnel, S2.pem. In addition, the server also has the client certificate, C1.crt for the first tunnel and C2.crt for the second tunnel. Likewise, the client has its certificates and private keys, C1.pem for the first tunnel and C2.pem for the second tunnel. The client also has the server certificate, S1.crt for the first tunnel and S2.crt for the second tunnel. Once we have presented the data from the example scenario, we can present the MultiTLS commands

to establish communication with two layers of protection. The commands required to establish this communication are:

1. Server-side: `multitls -s 11444 2 S1.pem C1.crt S2.pem C2.crt`

2. Client-side: `multitls -c 11444 2 192.168.1.119 C1.pem S1.crt C2.pem S2.crt`

It is essential that the certificates and keys are appropriate for the cipher suites used. In this case as we are using the default cipher suites of MultiTLS then the certificates of the files S1.pem, S1.crt, C1.pem and C1.crt must take into account that the first tunnel will use ECDHE for key exchange and ECDSA for authentication and the certificates in the S2.pem, S2.crt, C2.pem, and C2.crt files must contain RSA certificates.

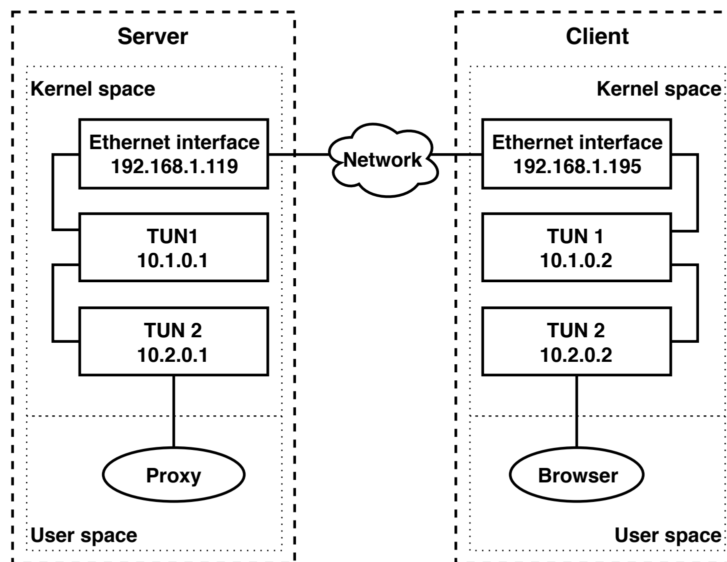

Figure 3.3: MultiTLS design with $k = 2$ to be used to establish communication between a browser and a proxy on different machines.

3.4 Implementing the tunnels

After presenting the MultiTLS commands, we can present how MultiTLS is implemented and how to create the tunnels. MultiTLS has as dependencies the socat version 1.7.3.2 [63] and OpenSSL version 1.1.0g tools.

The execution of commands provided by MultiTLS allows the creation of TUN interfaces and create tunnel which encapsulates a TLS channel between them as explained in Section 3.1. Figure 3.3 shows the scheme resulting from the execution of the two MultiTLS commands used in the example of Section 3.3, as well as using the communication created by MultiTLS to be used to establish communication between a browser and a proxy. It is worth remembering that

between the TUN1 interface of each host and between the TUN2 interface of each host there is a TLS channel.

The creation of these interfaces and the use of OpenSSL are responsible for the socat tool.

Socat is a command line based utility that establishes two bidirectional byte streams and transfers data between them. The use of socat can be applied to a wide variety of purposes since the streams can be constructed from a large set of different types of sources and sinks, also designated by address types, besides the multiple options that may be applied to streams.

A socat command has the following structure: socat [options] address1 address2, where [options] means that there may be zero or more options that modify the behavior of the program. The specification of the address1 and address2 consists of an address type keyword, for example, TCP4, TCP4-LISTEN, OPENSSL, OPENSSL-LISTEN, TUN; zero or more required address parameters separated by ':' from the keyword and each other; and zero or more address options separated by ','.

The life cycle of a socat process consists of four phases:

1. *Init phase*: the command line options are parsed and logging is initialized.

2. *Open phase*: Socat opens the first address and then the second address. These steps are usually blocking because connection or authentication requests must be completed before the next step is started.

3. *Transfer phase*: Socat watches both streams' read and write file descriptors, and, when data is available on one side and can be written to the other side, socat reads it and writes the data to the write file descriptor of the other stream, then continues waiting for more data in both directions.

4. *Closing phase*: This starts when one of the streams reaches EOF. Socat transfers the EOF condition to the other stream, i.e. tries to shutdown only its write stream, giving it a chance to terminate appropriately. Socat continues to transfer data in the other direction, but then closes all remaining channels and terminates.

As I mentioned previously, MultiTLS is a bash script and can be run in terminal as a command line program. In the beginning, the script starts by analyzing the arguments provided by the user. Afterward, these arguments are used to execute socat commands [64, 65]. MultiTLS creates k tunnels running k socat command on the server and k commands on the client. For the establishment of a tunnel using the socat commands, MultiTLS execute the following two commands, the first on the server side and the second on the client side:

- `socat openssl-listen:$port,cert=$cert,cafile=$cafile,cipher=$cipher \`
  `TUN:$ipTun/24,tun-name=$nameTun,up`
- `socat openssl-connect:$ipServer:$port,cert=$cert,cafile=$cafile,\`
  `cipher=$cipher TUN:$ipTun/24,tun-name=$nameTun`

In the first command, we have the \$port argument that represents the port where the socat will be listening, we have the \$cert, \$cafile and \$cipher arguments that have the same meaning as the cert, cafile and cipher arguments in the MultiTLS commands. The arguments \$ipTun and \$nameTUN are, respectively, the IP of the server in the TUN interface and the name of that, which is created through this command.

In the second command, we have the argument \$ipServer that represents the IP of the server, the argument \$port that represents the port of the server where the socat connects to establish the communication. We have the \$cert, \$cafile, and \$cipher arguments that have the same meaning as the cert, cafile, and cipher arguments in the MultiTLS commands. The arguments \$ipTun and \$nameTUN are, respectively, the IP of the client in the TUN interface and the name of that, which is created through this command.

MultiTLS by default assumes that the IP and names for the TUN interfaces are 10.\$k.1.\$i and TUN\$k, where \$k is the tunnel number, $1 \leq k \leq nTunnels$ and \$i has the value 1 if it is the server and 2 if it is the client.

After the establishment of the first tunnel, MultiTLS can create the second tunnel which is encapsulated by the first tunnel, using the previous socat commands in which the value of \$ipServer instead of being the real IP of the server is the IP of the TUN interface created on the server to establish the first tunnel, which as previously mentioned is 10.1.1.1, by default. In the same form, to create more tunnels, the IP of the last TUN interface created on the server side must be specified in the \$ipServer argument.

3.5 Summary

In this chapter, we introduced MultiTLS, a middleware that uses diversity and tunneling mechanisms to create multiple TLS channels and encapsulates each in other. We presented the TUN interfaces, the mechanism used by MultiTLS to encapsulate the TLS channels, each in another. We discuss the diversity of cryptographic mechanisms, presenting the metrics used to choose cipher suites that present the greatest diversity among them. We presented the arguments of the MultiTLS commands. Finally, we explain how MultiTLS is implemented, presenting the socat tool and its commands that are used by MultiTLS to create the secure tunnels.

Chapter 4

Evaluation

The use of diversity creates communication overhead, which results in performance costs. Each message sent is encrypted and signed $k-1$ times more than using a TLS implementation and every message received needs to be decrypted and checked $k-1$ times more. In addition, MultiTLS creates $k-1$ more TLS channels than using a TLS implementation and it encapsulates the k TLS channels into tunnels. In general, with this experimental evaluation, we want to know if the cost of replacing TLS channels with MultiTLS communication channels is acceptable.

The experimental evaluation aims to respond to questions such as the performance and cost of MultiTLS. For this, some experiments were performed and grouped into the following three topics:

1. MultiTLS performance;
2. comparison of MultiTLS with other approaches;
3. MultiTLS applied to a use case.

Section 4.1 shows the experiments that allow measuring the performance of MultiTLS. Section 4.2 compares MultiTLS with vtTLS and with an application that uses the DTLS protocol over a communication channel created by MultiTLS with $k=1$. Section 4.3 presents the results obtained in the use of MultiTLS to establish a communication channel between two hosts, where one was running a browser that was requesting some URLs to the proxy, which was in the other host, via MultiTLS. Finally, Section 4.4 presents the summary of this chapter.

4.1 Performance costs

In this section we want to answer the questions: what is the cost of adding more tunnels? What is the cost of encrypting messages? The answers to these questions allow us to understand

how MultiTLS performs. To answer these questions we used two virtual machines running on two different hosts, one playing the role of a server and the other of a client. Both virtual machines used 2VCPUs, 8GB of RAM, ran Ubuntu Xenial and using a gigabit network (Switch SMC8024L2).

In the first evaluation, we used the iperf3 tool [66], version 3.0.11. Iperf3 is a tool used to measure network performance. It has server and client functionality and can create data streams to measure the throughput between the two ends It supports the adjustment of several parameters related to timing and protocols. The iperf3 output presents the bandwidth, transmission time, and other parameters.

To answer the first question, we made our first experiment which consisted of using the iperf3 tool to measure 100 times the transmission time of 1 MB, 100 MB and 1 GB for each k, considering $k \leq 4$. The cipher suites used in this evaluation are the same ones that are defined by default in MultiTLS, this is TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256, TLS_RSA_WITH_AES_128_CCM_8, TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256, and TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384. The average and the standard deviation of transmission time of 1 MB, 100 MB and 1 GB for each value of k can be observed in Figure 4.1.

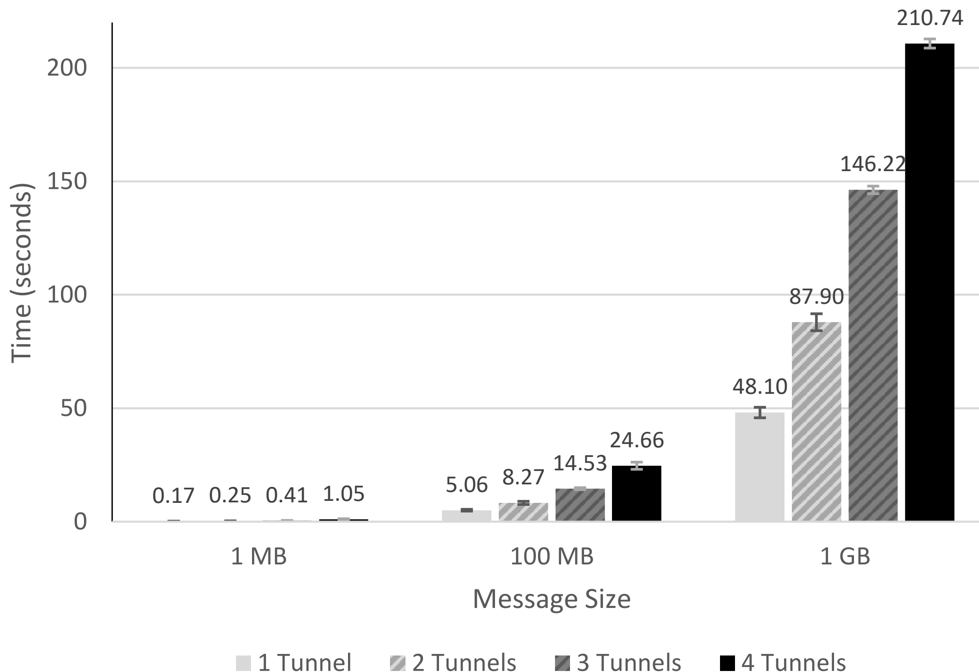

Figure 4.1: Comparison between the time it takes to send 1 MB, 100 MB and 1 GB messages in relation to the number of tunnels created.

Figure 4.2 shows for each message size the overhead of the transmission time for $k = 2$, $k = 3$ and $k = 4$ in relation to $k = 1$. Therefore, we can see that for $k = 2$ and $k = 3$ the cost of having added more tunnels increases as the size of the message to be transmitted also increases.

For $k = 4$ the cost of having added more channels decreased as the size of the message to be transmitted increased. We can also observe that the transmission time for k tunnels is less than k times the value of $k = 1$ for each message size, except for $k = 4$, where the overhead exceeds 4 times the value of $k = 1$ and for $k = 3$ in the 1GB transmission where the time is 3.04 times greater than for $k = 1$.

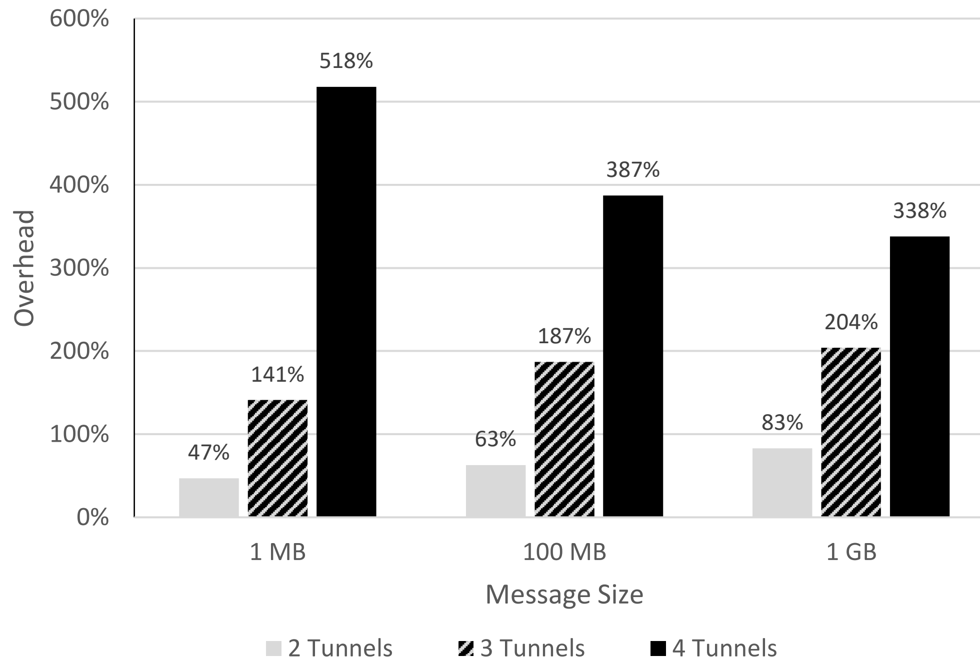

Figure 4.2: The overhead of adding more tunnels in relation to k = 1.

We can answer the first question that for $k = 2$ the performance of MultiTLS is good, since the time of sending messages with $k = 2$ is not more than twice the time of sending messages with $k = 1$. With 3 tunnels, i.e., $k = 3$, for the transfer of 1 GB, the performance of the MultiTLS is poor because the sending time is more than three times the time of $k = 1$, in contrast, to transfer 1 MB and 100 MB the performance is good since the sending time is less than three times the time of $k = 1$.

The second experiment aims to evaluate the cost of encrypting the communication messages. To do this, using the same vitual machines, we performed the same tests we did in the first experiment, but using the cipher suites TLS_ECDHE_ECDSA_WITH_NULL_SHA, TLS_RSA_WITH_NULL_SHA256, TLS_RSA_WITH_NULL_SHA and TLS_ECDHE_RSA_WITH_NULL_SHA. Therefore, the messages exchanged by the client and the server were not encrypted. For this experiment, we needed to use a previous version of OpenSSL, version 1.0.2g since we could not specify these cipher suites with version 1.1.0g. This experiment helps us realize the influence of encrypting the data in the total transmission time of messages with different sizes. Figure 4.3 shows the average and standard deviation of transmission time of 1 MB, 100 MB, and 1 GB for each value of k.

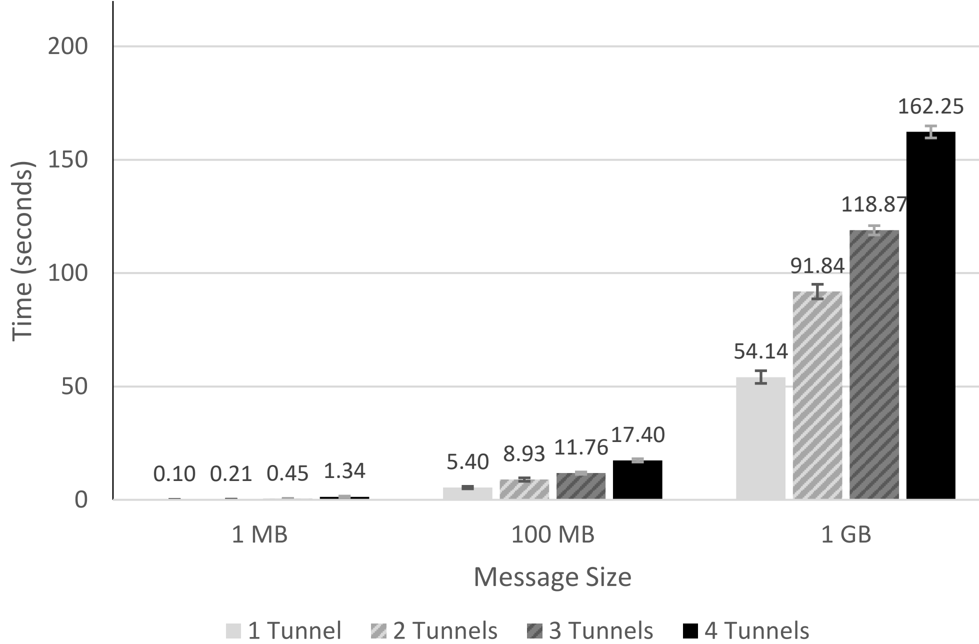

Figure 4.3: Comparison between the time it takes to send 1 MB, 100 MB and 1 GB messages in relation to the number of unencrypted tunnels.

As with the first experiment, for each message size, the transmission time increases as the number of tunnels increases. However, we verified that the transmission time of 1 MB for all values of k is greater than k times the time of $k = 1$. In the transfer of 100 MB and 1 GB with k tunnels, the transmission time does not exceed k times the value of $k = 1$.

Figure 4.4 shows the difference between the first and second experiment, for each message size and k. Strangely, we can see that, for certain message sizes and k, messages sent on the first experiment took less time than messages sent without encryption. This may be due to OpenSSL optimizations or some network congestion problem at the time of the experiments. However, we can observe that in these cases the average overhead is about -10%, whereas in cases where encrypted communications take longer than unencrypted communications, the average overhead is 35%. Overall, the overhead of encrypting the messages is 13%.

For all this, we can answer the second question that in general the time to encrypt the messages has a low impact on the sending time since the cost of this is 13%.

4.2 Comparisons with MultiTLS

One way to evaluate our tool is to compare with others that have the same goals. The purpose of this section is to compare the performance of MultiTLS with other tools and to know which of these approaches perform better?

For this purpose, using the same virtual machines that we used in previous experiments, we

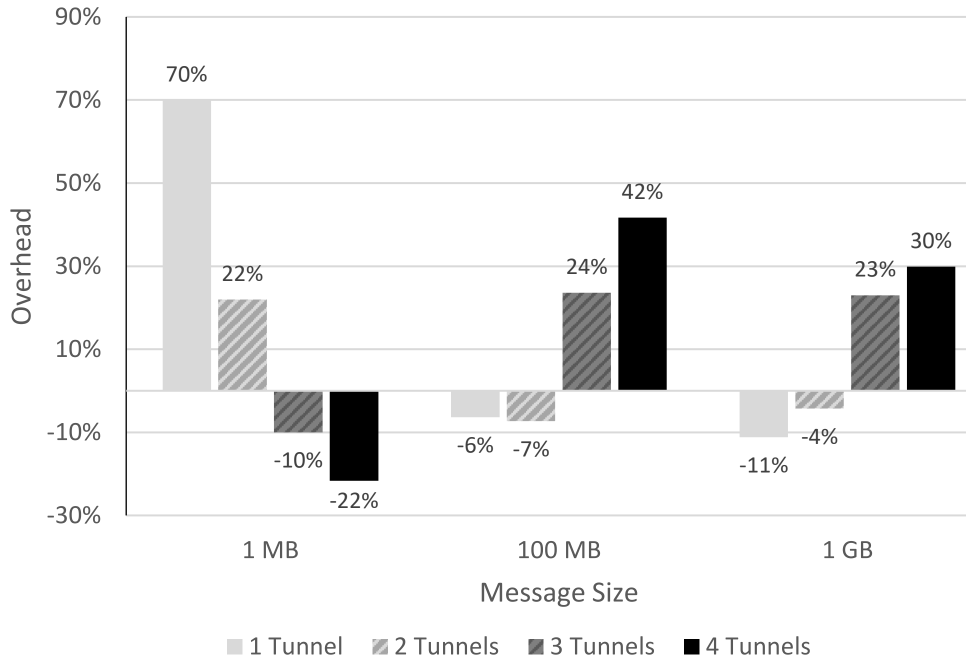

Figure 4.4: Overhead of the difference between first and second evaluation results.

use VTTLS to transfer three files each with the size of 1 MB, 100 MB and 1 GB. We ran 100 times the VTTLS for each of these files.

In addition to this experience, we also run a file transfer application using a Datagram Transport Layer Security (DTLS) [67] channel implemented through the GnuTLS library. This channel used the cipher suite TLS_RSA_AES_128_GCM_SHA256. This application ran over one tunnel created by MULTITLS. DTLS is a communication protocol that provides security, such as TLS, but for datagram-based applications. The purpose of using DTLS is to measure the performance of using a DTLS channel that uses UDP over a MULTITLS tunnel that uses a TCP channel, since with a MULTITLS communication with two tunnels or more we have TCP over TCP. We are interested in knowing that having TCP over TCP is harmful to the performance of MULTITLS since TCP over TCP usually has complications due to the meltdown effect [68].

We run this application 100 times for each of the files used in the previous experiment.

Besides the diversity of cipher suites used, this experience also shows that it is possible to have a diversity of TLS implementations if the application using MULTITLS uses a library other than OpenSSL.

Figure 4.5 allows us to compare the average of the results obtained from the two previous experiences with the averages of the results obtained in the first experiment with k = 2 once the two previous experiments use approaches in which the messages are encrypted twice such as MULTITLS with two tunnels. In addition, we can also observe the standard deviation in each column.

In this way, we can answer the question which of these approaches performs better, since Figure 4.5 allows us to see that, of the three approaches, vtTLS is the fastest and the DTLS channel approach is the slowest. The values of the MultiTLS results are closer to the results of the vtTLS than to the DTLS channel approach. However, the transfer time overhead of 1MB, 100MB and 1GB between vtTLS and MultiTLS are, respectively, 525%, 164% and 173%.

The DTLS channel approach does not have an expected performance for two reasons. The first is related to the fact that the client sends the size of the last fragment file that it received from the server, and secondly, the server only sends the next fragment after receiving the size of the last fragment sent by it.

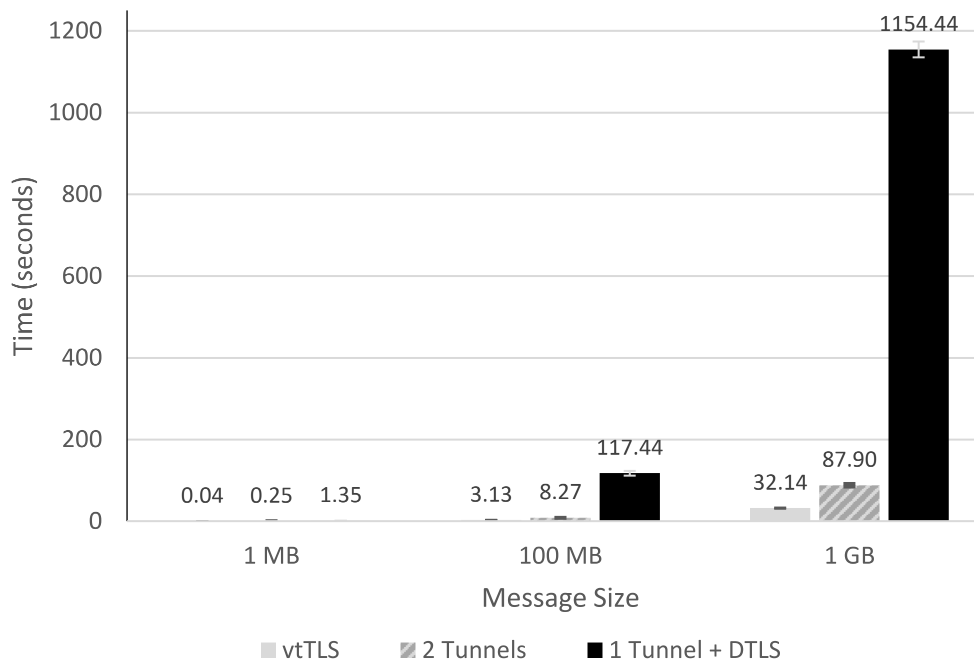

Figure 4.5: Time for sending messages with 1MB, 100MB and 1GB in size via vtTLS, two MultiTLS tunnels and one DTLS communication over one MultiTLS tunnel.

4.3 Use case

Although the use of MultiTLS presents a transfer time overhead in relation to vtTLS, we wanted to know what is the performance of MultiTLS applied in a more realistic use case.

To do this, we use MultiTLS to establish communication between a browser and a proxy, according to the scheme shown in Figure 4.6. To do this evaluation, we use two virtual machines, one ran the Squid proxy, version 3.5.12, on a computer with Intel Core i5 and 4 GB RAM and the other ran Google Chrome browser, version 66.0.3359.117, on a computer with Intel Core i7 and 8 GB RAM.

In this evaluation we tested four approaches: no proxy, use only the proxy, use the proxy

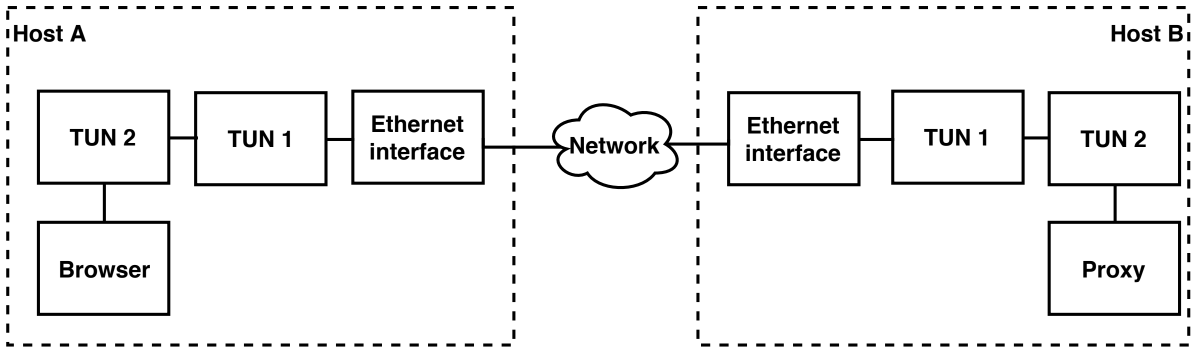

Figure 4.6: MultiTLS design with $k = 2$ to be used to establish communication between a browser and a proxy on different machines.

using one and two MultiTLS tunnels. These four approaches allow us to evaluate the cost of using MultiTLS.

The evaluation consisted of using the browser to request 30 times certain URLs from Amazon[1], Google[2], Safecloud[3], Técnico[4] and Youtube[5] websites for each approach and registered the value of the load event that appears on the network tab in the developer tools of the browser. The load event is fired when a resource and its dependent resources have finished loading. In addition to using the browser development tools to see the value of the load event, we also use to disable cache.

Figure 4.7 presents the average of the results obtained with the different approaches for each requested URL. We can observe that the use of MultiTLS in the communication between the browser and the proxy was insignificant, which leads us to conclude that MultiTLS is a tool with good performance in tasks like these that are recurrent in the day to day of the Internet users.

4.4 Summary

With this experiments, we conclude that the MultiTLS has a good performance for $k = 2$ and in some cases of $k = 3$. For $k = 4$ the performance of MultiTLS is unacceptable.

It is concluded that the cost of encrypting the data is insignificant since, on average, the communication channels that encrypted the data took 13% more to send the messages in relation to the time of sending the unencrypted messages.

Further observations we can derive from these experiments is that vtTLS performs better,

[1] https://www.amazon.com/
[2] https://www.google.com/
[3] http://www.safecloud-project.eu/
[4] https://tecnico.ulisboa.pt/pt/
[5] https://www.youtube.com/watch?v=oToaJE4s4z0

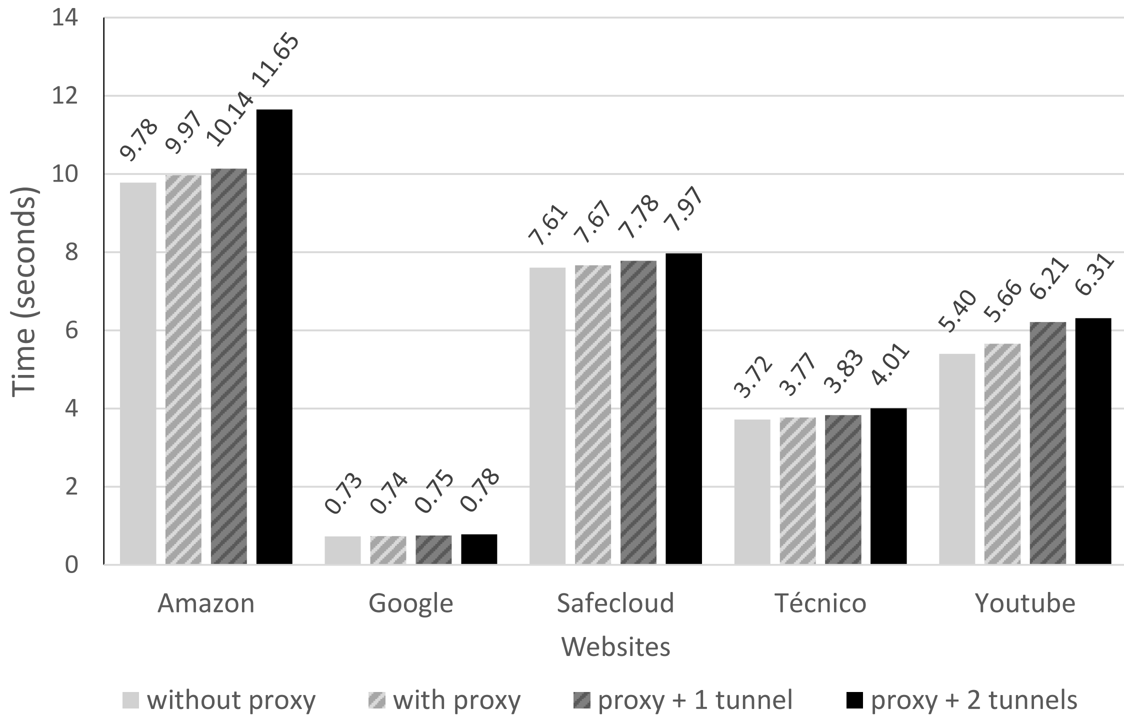

Figure 4.7: Time to load sites used the following 4 approaches: no proxy, with proxy, with proxy using communication created by MultiTLS with a tunnel and with two tunnels.

although it has some disadvantages compared to MultiTLS. It was observed that the time of sending messages through a MultiTLS communication channel with $k = 2$ (TCP over TCP) was faster than through a communication channel of an application that uses the DTLS over a MultiTLS communication channel with $k = 1$ (UDP over TPC).

Finally, we make a use case, in which a browser communicates with a proxy through the MultiTLS. The results of this experiment were very positive since the cost of using MultiTLS was not significant.

Chapter 5

Conclusions

MultiTLS is a middleware that allows the creation of a channel of communication through the encapsulation of several secure tunnels in others. It aims to increase security by using the diversity of cipher suites used by the tunnels so that if (k - 1) cipher suites become insecure, there is a secure tunnel that makes all communication secure.

In order to evaluate MultiTLS, several tests were executed with the intention of measuring its performance and cost. We compare MultiTLS with the protocol vtTLS and we conclude that although it performs poorly, MultiTLS has the advantages of not modifying any TLS implementation or any of its dependencies. In addition, MultiTLS can be used in a simple way by an application, such as communication between a browser and a proxy running on different hosts or by an application that allows us to create a TLS or DTLS channels. If these applications use a TLS library other than OpenSSL then diversity in TLS implementation is achieved, which makes communication more secure since the damage caused by implementation vulnerabilities in one of these implementations does not endanger communication.

5.1 Future Work

The use of diversity has grown greatly on communications security. Advancing this area will reduce vulnerabilities, making communication increasingly secure.

Although MultiTLS shows some progress in this area, there may be more work to do. Over the years certain cryptographic mechanisms become obsolescent and need to be replaced by secure ones. This requires that the study of measure the diversity of the different combinations of cipher suites be updated over the years.

In addition to the diversity of cipher suites, it would also be important to apply diversity in TLS implementations as it would allow mitigation of implementation vulnerabilities.

Bibliography

[1] M. Nadeau. State of Cybercrime 2017: Security events decline, but not the impact, July 2017. https://www.csoonline.com/article/3211491/security/state-of-cybercrime-2017-security-events-decline-but-not-the-impact.html#tk.cso_fsb, visited 2018-02-12.

[2] G. Greenwald and E. MacAskill. NSA Prism program taps in to user data of Apple, Google and others, June 2013. https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data, visited 2017-06-07.

[3] J. Fruhlinger. What is a cyber attack? Recent examples show disturbing trends, March 2018. https://www.csoonline.com/article/3237324/cyber-attacks-espionage/what-is-a-cyber-attack-recent-examples-show-disturbing-trends.html, visited 2018-04-23.

[4] Protocol Action: 'The Transport Layer Security (TLS) Protocol Version 1.3' to Proposed Standard (draft-ietf-tls-tls13-28.txt), March 2018. https://www.ietf.org/mail-archive/web/ietf-announce/current/msg17592.html, visited 2018-05-01.

[5] L. Bilge and T. Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. *In Proceedings of the ACM Conference on Computer and Communications Security*, pages 833–844, 2012.

[6] A. Joaquim. *vtTLS: A vulnerability-tolerant communication protocol.* PhD thesis, Instituto Superior Técnico, Universidade de Lisboa, 2016.

[7] A. Joaquim, M. L. Pardal, and M. Correia. Vulnerability-Tolerant Transport Layer Security. *21st International Conference on Principles of Distributed Systems (OPODIS)*, 2017.

[8] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, RFC Editor, August 2011.

[9] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, RFC Editor, January 1999.

[10] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol - Version 1.2. RFC 5246, RFC Editor, August 2008.

[11] P. Rogaway. Authenticated-Encryption with Associated-Data. September 2002.

[12] R. Shirey. Internet Security Glossary, Version 2. RFC 4949, RFC Editor, August 2007.

[13] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt. DROWN: Breaking TLS with SSLv2. *25th USENIX Security Symposium*, August 2016.

[14] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. *CRYPTO '98 Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–12, August 1998.

[15] OpenSSL - Cryptography and SSL/TLS Toolkit. https://www.openssl.org/, visited 2018-05-01.

[16] R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, RFC Editor, February 2012.

[17] M. Carvalho, J. Demott, R. Ford, and D. A. Wheeler. Heartbleed 101. *IEEE Security and Privacy*, 12:63–67, July/August 2014.

[18] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 1978.

[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. Public-Key Encryption. In *Handbook of Applied Cryptography*, chapter 8. CRC Press, 1996.

[20] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. Te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit rsa modulus. In *Advances in Cryptology – CRYPTO 2010*, pages 333–350. Springer Berlin Heidelberg, 2010.

[21] P. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1996.

[22] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[23] C. Rackoff and R. Simon, Daniel. Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack. *Advances in Cryptology — CRYPTO '91 SE - 35*, 576:433–444, 1992.

[24] H. Feistel. Data Encryption Standard (DES). *FIPS PUB 46 - 3*, 1999.

[25] R. C. Merkle and M. E. Hellman. On the security of multiple encryption. *Communications of the ACM*, 24(7):465–467, 1981.

[26] S. Lucks. Attacking Triple Encryption. *Fast Software Encryption, 5th International Workshop, FSE '98, Paris, France, March 23-25, 1998, Proceedings*, 1998.

[27] National Institute of Standards and Technology (NIST). Announcing the Advanced Encryption Standard (AES). 2001.

[28] A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique cryptanalysis of the full AES. In *Lecture Notes in Computer Science*, volume 7073 LNCS, 2011.

[29] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, RFC Editor, April 1992.

[30] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. *Advances in Cryptology – EUROCRYPT 2005*, 2005.

[31] E. Biham and A. Shamir. *Differential cryptanalysis of the data encryption standard.* Springer-Verlag Berlin, Heidelberg, 1993.

[32] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full SHA-1. pages 1–23, 2017.

[33] European Union Agency for Network and Information Security. *Algorithms, key size and parameters report - 2014.* 2014.

[34] D. Evans, a. Nguyen-Tuong, and J. Knight. Effectiveness of moving target defenses. *Moving Target Defense: An Asymmetric Approach to Cyber Security*, pages 81–100, 2011.

[35] M. Carvalho and R. Ford. Moving-target defenses for computer networks. *IEEE Security and Privacy*, 12(2), 2014.

[36] B. Littlewood and L. Strigini. Redundancy and Diversity in Security. *Computer Security ESORICS 2004*, pages 227–246, 2004.

[37] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 260–269, 2003.

[38] T. Roeder and F. B. Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems*, 28(2):1–54, 2010.

[39] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013*, 2013.

[40] R. Larson and L. Cockcroft. *CCSP : Cisco Certified Security Professional Certification.* McGraw-Hill/Osborne, 2003.

[41] W. Simpson. The Point-to-Point Protocol (PPP) (RFC 1661). RFC 1661, RFC Editor, July 1994.

[42] B. Lloyd and W. Simpson. PPP Authentication Protocols. RFC 1334, RFC Editor, October 1992.

[43] W. Simpson. PPP Challenge Handshake Authentication Protocol (CHAP). RFC 1994, RFC Editor, August 1996.

[44] G. Meyer. The PPP Encryption Control Protocol (ECP). RFC 1968, RFC Editor, June 1996.

[45] K. Sklower and G. Meyer. The PPP DES Encryption Protocol (DESE). RFC 1969, RFC Editor, June 1996.

[46] H. Kummert. The PPP Triple-DES Encryption Protocol (3DESE). RFC 2420, RFC Editor, September 1998.

[47] K. Sklower and G. Meyer. The PPP DES Encryption Protocol, Version 2 (DESE-bis). RFC 2419, RFC Editor, September 1998.

[48] K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, and G. Zorn. Point-to-Point Tunneling Protocol (PPTP). RFC 2637, RFC Editor, July 1999.

[49] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). RFC 1701, RFC Editor, October 1994.

[50] B. Schneier and Mudge. Cryptanalysis of Microsoft's point-to-point tunneling protocol (PPTP). *Proceedings of the 5th ACM Conference on Computer and Communications Security - CCS '98*, pages 132–141, 1998.

[51] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, and B. Palter. Layer Two Tunneling Protocol "L2TP" Status. RFC 2661, RFC Editor, August 1999.

[52] Defense Advanced Research Projects Agency (DARPA). Internet Protoco - DARPA Internet program - Protocol Specification. RFC 791, RFC Editor, September 1981.

[53] J. Reynolds and J. Postel. Assigned Numbers. RFC 1700, RFC Editor, October 1994.

[54] W. Simpson. IP in IP Tunneling. RFC 1853, RFC Editor, October 1995.

[55] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, RFC Editor, December 2005.

[56] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251, RFC Editor, January 2006.

[57] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protoco. RFC 4253, RFC Editor, January 2006.

[58] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252, RFC Editor, January 2006.

[59] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254, RFC Editor, January 2006.

[60] R. J. Carvalho. *Authentication Security through Diversity and Redundancy for Cloud Computing.* PhD thesis, Instituto Superior Técnico, Lisbon, Portugal, 2014.

[61] D. McGrew and D. Bailey. AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655, RFC Editor, July 2012.

[62] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, RFC Editor, May 2015.

[63] socat - Multipurpose relay (SOcket CAT), . http://www.dest-unreach.org/socat/doc/socat.html, visited 2018-05-01.

[64] Building TUN based virtual networks with socat, . http://www.dest-unreach.org/socat/doc/socat-tun.html, visited 2018-05-01.

[65] Securing Traffic Between two Socat Instances Using SSL, . http://www.dest-unreach.org/socat/doc/socat-openssltunnel.html, visited 2018-05-01.

[66] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/iperf-doc.php, visited 2018-05-01.

[67] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, RFC Editor, January 2012.

[68] O. Titz. Why TCP Over TCP Is A Bad Idea, April 2001. http://sites.inka.de/bigred/devel/tcp-tcp.html, visited 2018-06-17.