MULTITLS: Secure communication channel with diversity

Ricardo Manuel Mota de Moura
Instituto Superior Técnico, Universidade de Lisboa
Advisors: Professor Miguel Filipe Leitão Pardal and
Professor Miguel Nuno Dias Alves Pupo Correia

***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

I. 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. This means 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 [1].

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 developed 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 [2]. 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 [3], [4]. 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.

The remainder of this document is structured as follows. Section II presents background and related work. Section III presents the MULTITLS. Section IV presents the experimental evaluation. Section V presents the conclusion of the document.

II. BACKGROUND AND RELATED WORK

This chapter describes the protocol SSL/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 II-A describes the SSL/TLS protocol and the cryptographic mechanisms used by it. Section II-A1 presents vulnerabilities in the SSL/TLS protocol and cryptographic mechanisms. Section II-C refers to the advantages of using diversity in security and presents existing tunneling mechanisms.

A. SSL/TLS

This section provides basic information on TLS, discusses vulnerabilities in protocols and cryptographic mechanisms, and presents related work on diversity in security ans tunneling mechanisms.

1) The SSL/TLS protocol: The Secure Sockets Layer (SSL) [5] is a security protocol that provides secure communication channels between two entities - server and client. TLS protocol is structured into two layers: the TLS Record protocol and the TLS Handshake 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, the TLS Record protocol starts by fragmenting the message into blocks called TLSPlaintext. After the fragmentation step, each TLSPlaintext may be optionally compressed into a new block called TLSCompressed. Each TLSCompressed block is processed into a TLSCiphertext block by message authentication code (MAC) and encryption mechanisms. 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. 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.

The TLS Handshake protocol is used to establish or resume a secure session between server and client. A session is established in several steps, each corresponding to a different message and with a specific objective. Following the TLS Handshake protocol, the server and the client can exchange information through the secure communication channel.

B. TLS Vulnerabilities

Although the TLS protocol aims to establish secure communication channels, it may contain vulnerabilities making these channels insecure and susceptible to attacks. According to the Internet Security Glossary, Version 2 [6], vulnerabilities can be classified into three groups: design vulnerabilities, implementation vulnerabilities, and operation and management vulnerabilities. In this section, 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 vulnerability. The implementation vulnerabilities are related to failures that were created during the implementation phase of the protocol. To prove the importance of our work in increasing communications security, we present some vulnerabilities found in the TLS protocol and in some cryptographic algorithms used by it.

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.

2) Implementation vulnerabilities: In 2014, an implementation vulnerability was discovered in OpenSSL, called Heartbleed. The name of the vulnerability is related to an extension where a vulnerability appears, the heartbeat extension [7], which is an extension to the TLS protocol designed to enable a low-cost, keep-alive mechanism. 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 [8] 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 and dataset with size equal of the size specified in the received message minus the real size of the message.

3) Cryptographic vulnerabilities: This section presents some vulnerabilities in cryptographic mechanisms, specifically in some of the mechanisms supported by the TLS protocol.

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 [9] 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 [10]. 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 [11]. Although the use of RSA-1024 is currently discouraged, no factorization has yet been published.

Shor's algorithm [12] 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.

Vulnerabilities in symmetric cipher mechanisms: The Advanced Encryption Standard (AES) is an encryption algorithm created by Rijmen and Daemen [13]. 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.* [14] 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 datasets with a fixed length called hash values, checksums or simply hashes. A hash function is required to satisfy the following properties [10]:

- 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 [15] 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 [16]. They proved it through differential attacks, more specifically a modular differential attack. The differential cryptanalysis, introduced by E. Biham and A. Shamir [17], is a method which analyzes the effect of differences in input pairs on the differences of the resultant output pairs.

C. 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. Moving target defenses can classify into two groups: proactive and reactive. 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 [18]. 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. 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 better understand the approach followed.

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. 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 [19] is a mechanism that restores some measure of this independence by restarting each replica periodically with a newly generated and diverse executable.

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 [20] is an approach that applies these concepts as a way of solving problems related to reused code and code injection.

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.

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.

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 [21].

The Internet Protocol (IP) transmits block of data called datagrams from sources to destinations, which are hosts identified by addresses [22].

In the IP header of the packets there is a field, called Protocol, to identify the next level protocol [23]. In this field we can used the IP in IP Tunneling protocol.

In IP in IP Tunneling [24], 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.

IPsec [25] 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.

The Secure Shell Protocol (SSH) is a protocol for secure remote login and other secure network services over an insecure network [26]. SSH is typically used to log into a remote machine and execute commands, but it also supports tunneling.

SSH may have been divided into the following three layers:

- Transport layer protocol - it provides encryption, server authentication, and integrity protection [27];
- 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 [28];
- 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 [29].

These layers provide mechanisms that make SSH secure for tunneling.

III. 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 contributes to 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.

In the following sections, we will discuss the TUN interfaces which are the mechanism used by MULTITLS to encapsulate the various TLS channels. The study that enabled us to choose the combination of cipher suites supported by MULTITLS and that guarantee greater diversity, we present how to use MULTITLS to create the various TLS channels. Finally, we present how the middleware is implemented and with it creates the TLS channels.

A. 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 1 shows a practical example in which an application running on two different hosts communicates through 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.

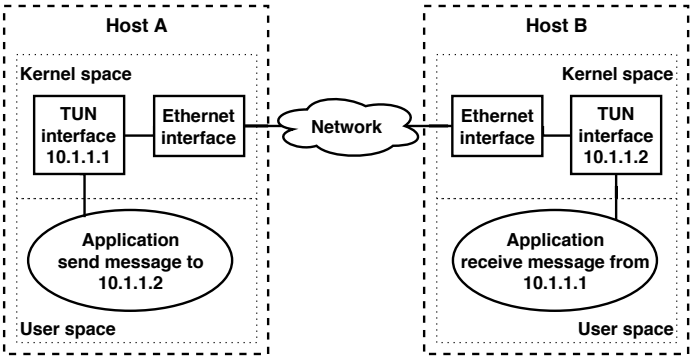

Fig. 1: Example of using TUN interfaces

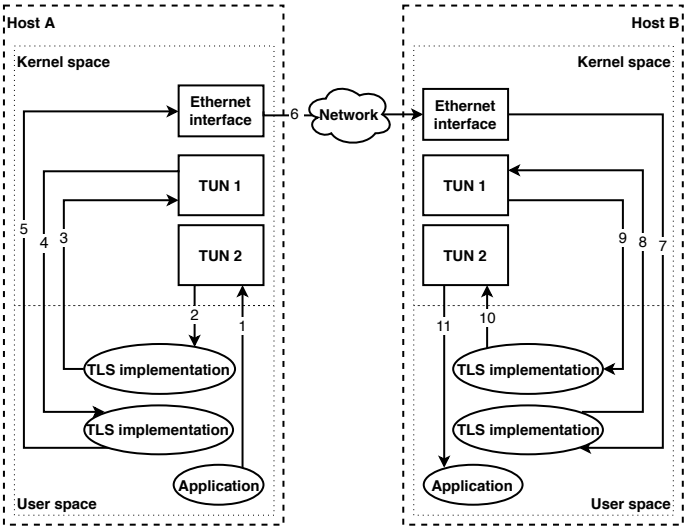

Fig. 2: MULTITLS design with $k = 2$ and the flow of sending messages from one application to another on different hosts

Figure 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.

B. Combining Diverse Cipher Suites

In MULTITLS, we are interested in having 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 [30], 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; and DHE for key exchange and RSA for authentication. In order to avoid that the key exchange and authentication algorithms are repeated consecutively, 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. 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 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. 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, 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) 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. 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 and 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.

C. 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. The commands available through MULTITLS are:

- multitls -s port nTunnels [cert cafile cipher]
- 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. If not specified cipher suites will be used by default. The arguments between brackets must be specified the number of times that the value of the nTunnels argument has.

D. 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 tools socat version 1.7.3.2 [31] and OpenSSL version 1.1.0g.

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 III-A. Figure 2 shows the scheme resulting from the execution of the two MULTITLS commands present in Section III-C. 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, OPENSSL, TUN; zero or more required address parameters separated by ':' from the keyword and each other; and zero or more address options separated by ','.

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 [32], [33]. 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.

IV. EVALUATION

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.

We will start by showing the assessments that we performed to measure the performance of MULTITLS. Next, we will observe the MULTITLS comparisons with VTTLS and with an application that uses the DTLS protocol on a tunnel created by MULTITLS. Finally, we will observe the results obtained in the use of MULTITLS in the communication of a browser with a proxy.

A. 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, 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. 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 3.

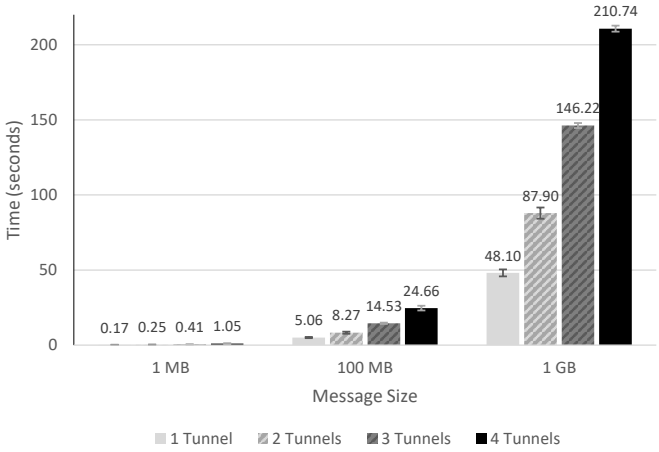

Fig. 3: Comparison between the time it takes to send 1 MB, 100 MB and 1 GB messages in relation to the number of tunnels created.

Figure 4 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 $k = 1$.

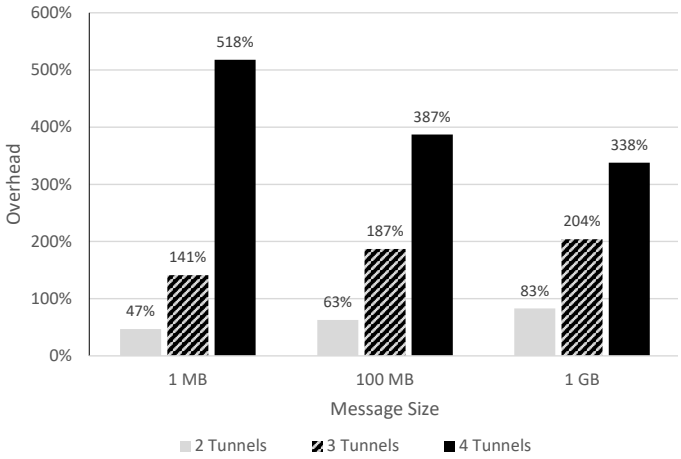

Fig. 4: The overhead of adding more tunnels in relation to k = 1.

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, however changing the cipher suites by default from MULTITLS to 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 5 shows the average and standard deviation of transmission time of 1 MB, 100 MB, and 1 GB for each value of k.

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 $k = 1$.

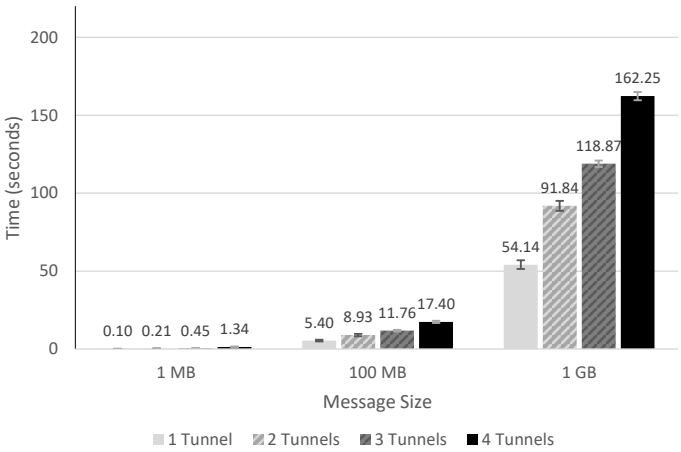

Fig. 5: Comparison between the time it takes to send 1 MB, 100 MB and 1 GB messages in relation to the number of unencrypted tunnels.

Figure 6 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%.

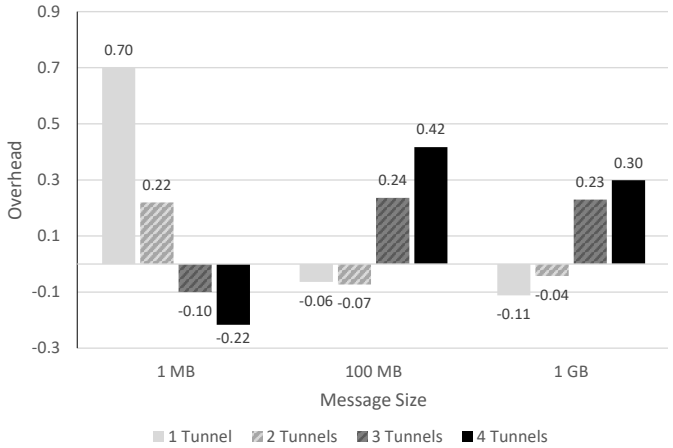

Fig. 6: Difference between first and second evaluation results.

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%.

B. *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 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) [34] 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 detrimental to the performance of MULTITLS since TCP over TCP usually has complications due to the meltdown effect [35]. 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 7 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 7 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.

C. 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, based on the scheme shown in Figure 2. 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.

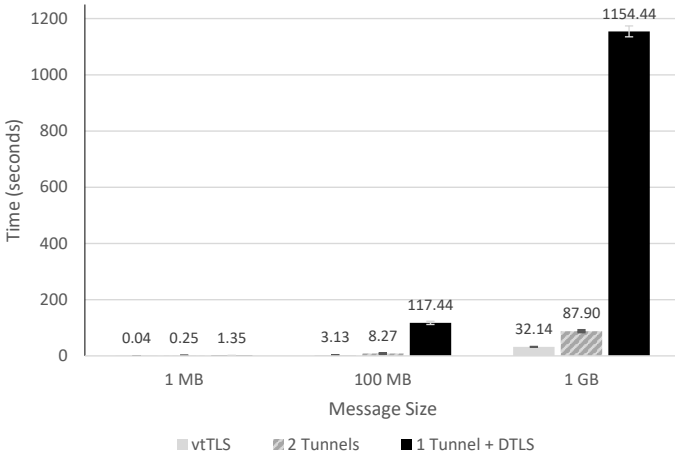

Fig. 7: Time for sending messages with 1MB, 100MB and 1GB in size via VTTLS, two MULTITLS tunnels and one DTLS communication over one MULTITLS tunnel.

In this evaluation we tested four approaches: no proxy, use only the proxy, use the proxy 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], Tcnico[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.

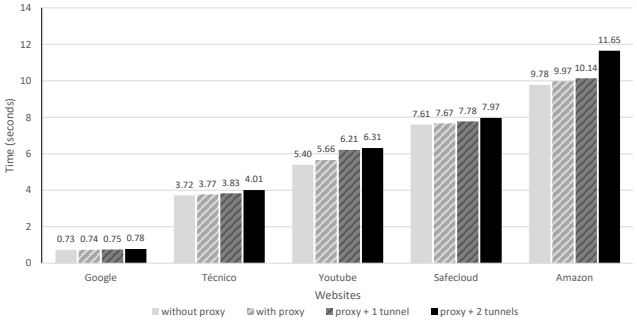

Fig. 8: Time to load sites used the following 4 approaches: no proxy, with proxy, with proxy using communication created by MULTITLS with a tunnel and with two tunnels.

Figure 8 presents the average of the results obtained with the different approaches for each requested URL. We can observe

[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

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.

V. CONCLUSION

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.

A. 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.

REFERENCES

[1] "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.

[2] 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*, pp. 833–844, 2012.

[3] A. Joaquim, "vtTLS: A vulnerability-tolerant communication protocol," Ph.D. dissertation, Instituto Superior Técnico, Universidade de Lisboa, 2016.

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

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

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

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

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

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

[10] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, "Public-Key Encryption," in *Handbook of Applied Cryptography*. CRC Press, 1996, ch. 8.

[11] 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*. Springer Berlin Heidelberg, 2010, pp. 333–350.

[12] P. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1996.

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

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

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

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

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

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

[19] T. Roeder and F. B. Schneider, "Proactive obfuscation," *ACM Transactions on Computer Systems*, vol. 28, no. 2, pp. 1–54, 2010.

[20] 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.

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

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

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

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

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

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

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

[28] ——, "The Secure Shell (SSH) Authentication Protocol," RFC Editor, RFC 4252, January 2006.

[29] ——, "The Secure Shell (SSH) Connection Protocol," RFC Editor, RFC 4254, January 2006.

[30] R. J. Carvalho, "Authentication Security through Diversity and Redundancy for Cloud Computing," Ph.D. dissertation, Instituto Superior Tcnico, Lisbon, Portugal, 2014.

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

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

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

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

[35] 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.