MultiTLS: using multiple and diverse ciphers for stronger secure channels 

Ricardo Moura[a], Ricardo Lopes[a], David R. Matos[a], Miguel L. Pardal[a], Miguel Correia[a]

[a]*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Rua Alves Redol, 9, Lisboa, 1000-029, Lisbon, Portugal*

Abstract

Computer communication is at the foundation of how the modern world works, connecting people and machines over public infrastructure. For this reason, communication is exposed to attacks, either by passive listening, or by active interference in the communication. Security protocols like TLS (Transport Layer Security) play a crucial role in ensuring the confidentiality, integrity, and authenticity of the communication. However, like in all technologies, there may be flaws in the design, implementation, or cryptography of TLS that compromise the security of the communication channel. Remediation of such vulnerabilities takes time, leaving valuable services exposed to potential attacks.

In this article, we present MultiTLS, a middleware based on cipher diversity and network tunneling that enables secure communication even when new vulnerabilities are discovered. MultiTLS creates a secure communication tunnel through the encapsulation of k TLS channels, where each one uses a different cipher suite. This approach allows the communication channel to remain protected, even when $k - 1$ cipher suites become vulnerable, because of the remaining cipher suite. The diversity of cipher suites tolerates cryptography faults. We evaluated the implementation of MultiTLS and concluded that it is easy to use and to maintain up-to-date, since it does not require code changes to any of its dependencies. We also evaluated its performance in practical use cases and proved that it is viable an useful for various personal and corporate contexts using Internet communications.

Keywords: Secure communication channels, Transport Layer Security, Vulnerability-tolerance, Security through Diversity, Tunneling, Virtual Private Network

1. Introduction

We live in an increasingly digital age where a large part of services, such as banking, shopping, and healthcare are accessed through the public Internet. There have been many cyberattacks that caused increased losses and damage to businesses and Internet users (Nadeau, 2017). This means that, nowadays, the use of cryptography-based secure communication protocols are a fundamental component of distributed systems and digital business. They allow entities to exchange messages through a trusted communication channel over the untrusted public Internet. These channels aim to guarantee the following three properties: *confidentiality*: ensure that only the intended receiver is able to read the message; *integrity*: ensure that messages cannot be changed without the receiver detecting it; *authenticity*: ensure that the identity of the sender and receiver can be verified.

Transport Layer Security (TLS) is one of the most widely used secure communication protocols. The protocol allows server/client applications to communicate over a channel that is designed to prevent eavesdropping, tampering, and message forgery. The most recent version is TLS 1.3 (Rescorla, 2018). This protocol first appeared under the name *Secure Sockets Layer* (SSL). It is the final 'S' in HTTPS that stands for 'Secure' and is visually perceived by end-users as the "padlock" in the web browser that signifies to them that the communication is secure. 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. 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 vulnerability to CBC block chaining attacks and supporting more block encryption modes for use with AES (Advanced Encryption Standard). In 2018, TLS 1.3 was approved by the Internet Engineering Task Force (IETF), becoming the current standard for secure connections, even though version 1.2 is still the most widely used.

1.1. Secure Channel Vulnerabilities

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 cryptographic mechanisms vulnerabilities. We will discuss vulnerabilities at length in Section 2.2.

When a new vulnerability is found, it will take significant time for it to be fixed (Bilge and Dumitras, 2012). First, the owners of the software or hardware need to determine at which level is the vulnerability: is it a design or implementation flaw? Once the diagnosis is complete, remediation can start. After some time, new versions and patches are available, but they still need to be provided to third-parties that, in turn, will plan the most appropriate time for installation. Finally, and over time, the patches are applied and the vulnerability is fixed in existing deployments. However, there will be some deployments that are never updated. During this whole time, communication channels are exposed to attackers.

1.2. Security through Diversity

This work explores diversity in communication protocols by using multiple cipher suites. These suites are used for defining: a key exchange algorithm, an authentication mechanism, an encryption mechanism, and a message integrity protection. To implement diversity, we intended to use existing libraries and tools without modification, to be able to always benefit from the latest and most secure versions of them.

We developed MultiTLS, a middleware that obtains diversity by leveraging tunneling mechanisms. In our implementation, we used *socat*[1], a tunneling software, and OpenSSL[2], a TLS implementation, to create multiple TLS channels and encapsulate each one in another. The source code for MultiTLS is publicly available[3] with an open-source license.

MultiTLS can be run as a command in the Linux shell and is configured with a parameter k, called the *diversity factor* ($k > 1$). This parameter specifies the number of TLS channels to be created and consequently the

[1] `http://www.dest-unreach.org/socat/`
[2] `https://www.openssl.org/`
[3] `https://github.com/inesc-id/MultiTLS`

number of cipher suites to be used. $k = 1$ is equivalent to a single TLS channel. The cipher suites used for multiple TLS channels are different from each other to mitigate the vulnerabilities that may be found in each one. This approach can provide security even in the presence of zero-day vulnerabilities which can not be prevented as they are unknown (Bilge and Dumitras, 2012).

The communication channel created by MultiTLS has multiple layers of protection, so that if $k - 1$ of the used cipher suites are vulnerable, communications will remain secure, since there is at least one cipher suite that guarantees the security of communications, i.e, the confidentiality, integrity, and authenticity properties.

MultiTLS is an improvement over a previous work, vtTLS (Joaquim et al., 2017), a vulnerability-tolerant communication protocol also based on diversity and redundancy of cryptographic mechanisms to provide a secure communication channel. However, vtTLS had some problems with software maintenance because it modified an existing TLS implementation. On the other hand, MultiTLS can always incorporate the latest updates and security fixes because it supports the latest versions of TLS transparently.

1.3. Overview

The remainder of this document is structured as follows. Section 2 presents background and related work. Section 3 presents MultiTLS in detail. Section 4 presents the experimental evaluation. Finally, Section 5 presents the conclusions.

2. Background and Related Work

In this section, we describe the SSL/TLS protocol and its basic design, presents some of the most important vulnerabilities in the TLS protocol and in the cryptographic mechanisms used by it. We also discuss related work on approaches to achieve security through diversity. Finally, we summarize existing network tunneling mechanisms.

2.1. The SSL/TLS Protocol

The SSL (Secure Sockets Layer) (Freier et al., 2011) / TLS (Transport Layer Security)(Rescorla, 2018) is a security protocol that provides secure communication channels between two entities, server and client. The protocol is structured in two layers: the TLS Record protocol and the TLS Handshake

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

In regard to sending messages, the 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: a session identifier (chosen by the server), the certificates (X509 standard), the compression algorithm used to originate the TLS Compressed blocks in the Record Protocol, the specifications cipher (MAC and cipher algorithm used in the Record Protocol to generate the `TLSCiphertext`), a master secret (shared between the client and the server) and the "is resumable" flag that indicates whether the session can be used to initiate new connections. The Change Cipher Spec Protocol consists of a message encrypted and compressed according to the current state of the connection, to signal a change in the set of negotiated ciphers. The Alert Protocol sends an alert message that, depending on the severity, can be of the warning or fatal type (warning/fatal). These messages are encrypted and compressed based on the current connection status. Following a successful handshake, the server and the client can exchange information through the established secure communication channel.

2.2. TLS Vulnerabilities

Although the goal of the TLS protocol is to establish a secure communication channel, it may still have unknown vulnerabilities making it insecure and susceptible to attacks. According to the Internet Security Glossary, Version 2 (Shirey, 2007), vulnerabilities can be classified into three groups: design vulnerabilities, implementation vulnerabilities, and operation and management vulnerabilities. In this work, we focus only on the first two groups

of vulnerabilities. The design vulnerabilities refer to protocol specification failures and releasing a new version or update is the only way to fix this kind of 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 of the cryptographic algorithms used by it.

2.2.1. Design vulnerabilities

An example of an attack that exploits a design vulnerability is CRIME (Compression Ratio Info-leak Made Easy) (Rizzo and Duong, 2012). 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.

2.2.2. Implementation vulnerabilities

In 2014, an implementation vulnerability was discovered in OpenSSL, called Heartbleed. The name of the vulnerability is related to a code extension where the vulnerability was found: the Heartbeat extension (Seggelmann et al., 2012), 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 obtains this message, it returns the received payload.

The Heartbleed vulnerability (Carvalho et al., 2014) is a buffer over-read vulnerability that happens when the sender sends a message that specifies a payload size bigger than the real size of the payload. 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 to the size specified in the received message minus the real size of the message. This allows potential attackers to read memory contents that should have been kept private.

There are also vulnerabilities in the underlying cryptographic mechanisms used by the TLS protocol. Our solution uses diverse cipher suites as a form to increase security. For this, it is necessary to study the vulnerabilities in the cryptographic mechanisms to know which cipher suites are more secure and which can be used.

2.2.3. Vulnerabilities in asymmetric cipher mechanisms

RSA (Rivest et al., 1978) proposed by Rivest, Shamir and Adleman, in 1978, is an asymmetric cryptographic algorithm used to cipher and sign messages. The security of RSA is based on two problems: integer factorization problem and the RSA problem itself (Menezes et al., 1996). 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 because Kleinjung et al. (2010) have been able to factor a number with 768 bits, equivalent to a number with 232 digits. Although the use of RSA-1024 is currently discouraged, no factorization has yet been published. Shor's algorithm (Shor, 1996) uses a theoretical quantum computer to factorize integers in polynomial time, making the integer factorization problem easy to solve. However, this problem will only exist when quantum computers are practical and available.

2.2.4. Vulnerabilities in symmetric cipher mechanisms

The Advanced Encryption Standard (AES) is an encryption algorithm created by Rijmen and Daemen, and standardized by the NIST (2001). 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. Bogdanov et al. (2011) published a 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. Despite this attack and others, AES is still considered a secure encryption mechanism.

2.2.5. 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 or checksums. A hash function is required to satisfy the following properties (Menezes et al., 1996):

- Easy to compute the hash value for any given message;
- Preimage resistance: it is infeasible to generate a message that has a given hash value;
- Second preimage resistance: it is infeasible to modify a message without changing the hash value;
- Collision resistance: it is 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 it useful for data integrity checks and message authentication. Note that it is impossible to have a unique identity once the message is compressed, allowing attackers to break the collision resistance property.

MD5 (Rivest, 1992) is a hash function that produces a 128 bit hash. MD5 was proved not to be collision resistant by Wang and Yu (2005), through differential attacks. Differential cryptanalysis, introduced by Biham and Shamir (1993), analyzes the differences in input pairs on the differences of the resultant output pairs.

2.3. Achieving security through diversity

In this work we aim to achieve 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 be classified 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 between themselves (Littlewood and Strigini, 2004). With diverse versions, one hopes that any faults they contain will be different and show different failure behavior.

In MultiTLS we allow diverse ciphers to be combined arbitrarily, because of the tunnelling approach, enabling moving target defense in the use of ciphers for secure communication.

2.3.1. Vulnerability-Tolerant TLS

The use of diversity for added security in a communication channel was previously used by Joaquim et al. (2017) in vtTLS. It also uses the diversity approach to solve the limitation of TLS having only one cipher suite negotiated between server and client. In these cases, if one of the cryptographic mechanisms of the cipher suite becomes insecure, the communication channels using this cipher suite may become vulnerable. It uses 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 successfully implemented as a fork of OpenSSL version 1.0.2g, but moving to a newer version of the library requires implementing the diversity features again. And again, for all future versions. Our solution, MultiTLS, is similar to this approach but it does not modify implementations of the libraries and tools, and only their public interfaces are used. Because of this, this solution is able to use the latest and most secure versions of the software.

2.3.2. Tunneling

The term *tunneling* describes a process of encapsulating entire data packets as the payload within other packets, which are handled properly by the network on both endpoints (Larson and Cockcroft, 2003). This characteristic in this type of protocol makes it possible to send data between two private networks, using a public network infrastructure.

A communication tunnel is an essential component of a VPN, short for Virtual Private Network, a technology to ensure that sensitive data can be transmitted securely, preventing unauthorized persons from having access

to this information. When talking about VPN there are several types to consider (Khanvilkar and Khokhar, 2004): Machine-to-Machine, Machine-to-Network, and Network-to-Network.

For the tunnel connection to be successfully established, it is essential that both parties understand and use the same protocol. The Internet Protocol (IP) transmits block of data called datagrams from sources to destinations, which are hosts identified by addresses, as defined by Postel (1981). In the IP header of the packets there is a field, called Protocol, to identify the next level protocol (Reynolds and Postel, 1994). In this field we can used the "IP in IP" Tunneling protocol. In IP Tunneling (Estrin et al., 1995), 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 (Kent and Seo, 2005) 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 subprotocols 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 (T. Ylonen and C. Lonvick, 2006). SSH is typically used to log into a remote machine and execute commands, but it also supports tunneling. SSH is structured in three layers that provide the mechanisms that make SSH secure for tunneling:

- Transport: provides encryption, server authentication, and integrity protection (Ylonen and Lonvick, 2006c);
- Authentication: runs on top of the Transport layer and provides ways to authenticate the client to the server (Ylonen and Lonvick, 2006a);

- Connection: also runs on top of the Transport layer and specifies a mechanism to multiplex multiple channels over the underlying confidentiality and authentication transport (Ylonen and Lonvick, 2006b).

3. MultiTLS

MultiTLS provides secure communication channels with multiple layers through *tunneling* of TLS channels within each other. MultiTLS 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. In practical terms, we expect the value of k to range from 1 to 3. A value of 1 represents a secure tunnel with baseline encryption, while adding more different ciphers enhances the ability to tolerate vulnerabilities. Values beyond 3 are not very likely due to the limited availability of diverse cipher suites and the accumulated impact on performance, as discussed in Section 4.2.

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 creating k TLS channels and encapsulate one into another without changing the implementations of the used tools. This approach is an advantage over vtTLS, since it does not require changes to the implementation of TLS.

When a vulnerability is discovered, its remediation is not instantaneous, as it needs to be understood, the software needs to be fixed, and the patches need to be distributed across many deployments. In the meantime, attackers can target the valuable services that are exposed. MultiTLS provides enhanced flexibility in addressing the issue. Unlike a single TLS channel that necessitates immediate attention, MultiTLS with a value of $k > 1$ allows operations to continue while the vulnerability and its impact are handled.

Updates can be scheduled at a later time to minimize disruptions, offering a more flexible and efficient approach to resolving security problems.

In the following sections, we will discuss use cases, followed by the design and implementation of MultiTLS.

3.1. Use cases

MultiTLS can be used to add security to a communication channel without protection or to reinforce the security of existing but weak protection. To contextualize the use of MultiTLS in practical scenarios, four case studies were defined where the use of the tool can offer security advantages:

1. Secure communication between two organization networks;
2. Secure communication between two cloud solutions;
3. Secure communication between the employee's device and the organization's network;
4. Secure communication between legacy applications.

In the case of secure communication between networks, it will be necessary to configure the MultiTLS tool in both networks, which will work as a reverse proxy. It will allow secure connections through multiple encrypted TLS channels, which reinforce the level of security between two areas of operation of an organization, e.g. two buildings in different locations.

In the case study of secure communication between two cloud solutions, MultiTLS reinforces security in the communication between two cloud solutions, whether from the same provider or from different providers. Here too, it will be necessary to configure a machine in each cloud solution that will serve as a reverse proxy for the remaining assets.

The case study of secure communication between the employee's device and the organization's network, is intended to represent a scenario when the employee is outside the organization's network, for example, working from home or from an hotel. It will be the responsibility of the organization to guarantee the availability of the service (MultiTLS as server), accepting connection requests. On the employee's side, she must configure the tool in client mode and establish the connection with the server.

The last case, secure communication between legacy applications, is focused on existing applications, possibly with obsolete technologies but that still play a critical business function. They may even have known security vulnerabilities. It will be necessary to configure the MultiTLS tool on

the machines where these applications are located. The use of the tool allows communication to be carried out securely by encapsulating the legacy application message through recent and secure cryptographic protocols. A specific example would be the interconnection of an application server with a database server that does not support a recent TLS protocol version.

The first two case studies correspond to network-to-network VPNs, the third case study pertains to a host-to-network VPN, and, finally, the fourth use case corresponds to a machine-to-machine VPN.

3.2. Design

To encapsulate a TLS channel in another TLS channel, we use network tunnel interfaces (abbreviated as TUN interfaces). This mechanism is a feature offered by some operating systems, namely, Linux. Unlike common network interfaces, TUN interfaces do not have physical hardware components, that is, they are virtual network interfaces implemented and managed by the kernel itself. TUN is a virtual point-to-point network device. Its driver was designed with 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 network stack of the operating system. 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 network hosts communicate through TUN interfaces.

We create an encapsulation of several tunnels by creating TUN interfaces through others created previously. For each of these interfaces, we can use different TLS implementations running in user space that allow creating a TLS channel that is encapsulated by the tunnel used by the hosts.

Figure 2 presents the architecture of MultiTLS for host communicating over the network with $k = 2$. This parameter configuration allows communication over two tunnels, where the tunnel between the TUN1 interfaces encapsulates the tunnel between the TUN2 interfaces. In addition, we can see processes that we designate as "TLS implementation". These processes serve the purpose of setting up and overseeing the TLS channel for each tunnel, operating as client on one side, or as server on the other side.

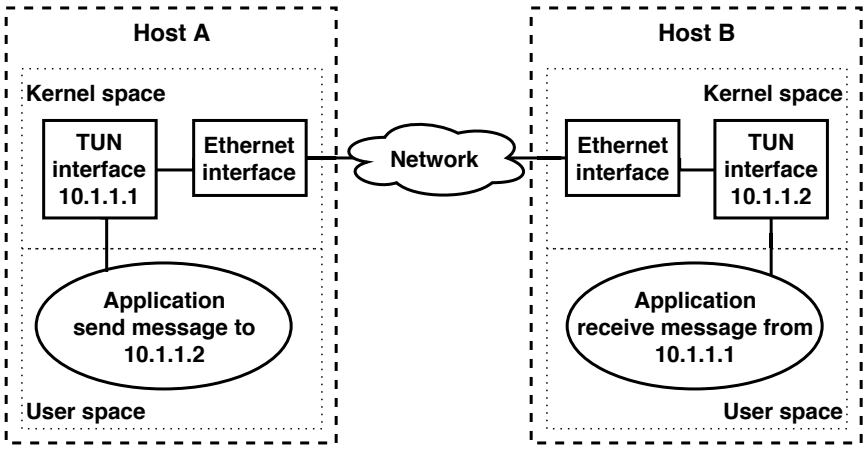

Figure 1: Example of using TUN interfaces.

3.3. Combining Diverse Cipher Suites

In MultiTLS, we are interested in having the maximum possible diversity of cryptographic mechanisms, because we want to avoid common vulnerabilities. Evaluating the diversity among cryptographic mechanisms is not trivial. For this purpose, we based our analysis on work by Carvalho (2014) regarding heuristics to compare diversity among different cryptographic mechanisms. In our work, we focused on searching for the combination of four cipher suites that guarantee greater diversity and are supported by TLS 1.2 from the OpenSSL 1.1.0g implementation.

We began by evaluating the diversity of public key mechanisms. In this case, we observed 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 forward 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.

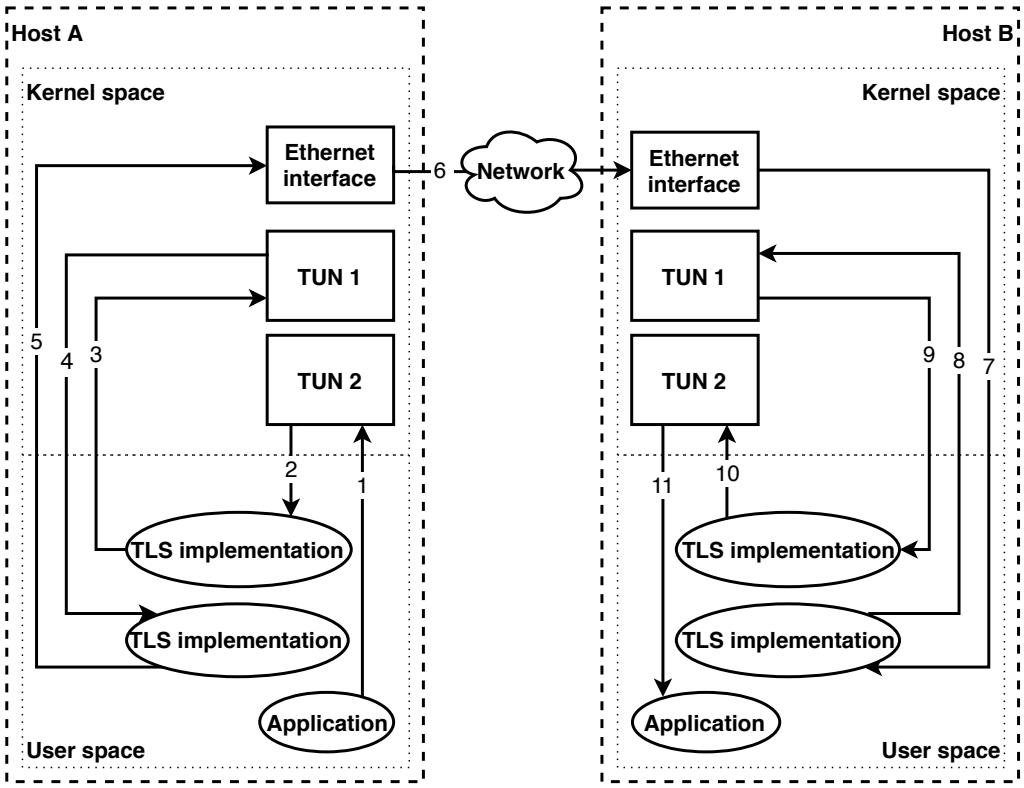

Figure 2: MULTITLS design with $k = 2$ and the flow of sending messages from one application to another on different hosts.

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. 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 described in research by Carvalho (2014). We concluded 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 operation modes, 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 hash which is a one-time message 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;
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384.

If the MultiTLS user selects 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.

3.4. Interception resistance

A man-in-the-middle (MITM) attack occurs when an attacker intercepts and potentially manipulates the communication between two parties, allowing the attacker to eavesdrop or alter the messages, or to impersonate one of the parties. Although TLS is designed to safeguard against MITM attacks,

vulnerabilities can still arise in certain implementations or configurations, making such attacks possible. MultiTLS introduces multiple intermediate protection levels, and so it increases the difficulty for attackers attempting to carry out interception attacks.

3.5. Running MultiTLS

MultiTLS was implemented as a script in Bash language and can be run as a shell command on Linux. Before presenting how MultiTLS creates the secure tunnels, we will first introduce the commands that allow us to create them:

- `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: the path to the file with its certificate and private key in the *cert* argument and the path to the file that contains the peer certificate in the *cafile* argument. The *cipher* argument lets us specify one or more cipher suites. If cipher suites are not specified, the default ones will be used. The arguments between brackets must be specified as many times as the value of the *nTunnels* argument because each tunnel will use a set of keys and ciphers.

3.6. Implementing the tunnels

The execution of commands provided by MultiTLS allows the creation of TUN interfaces and the creation of the tunnel that encapsulates a TLS channel, as explained in Section 3.2. Figure 2 shows the scheme resulting from the execution of the two MultiTLS commands as shown in Section 3.5.

MultiTLS depends on the socat version 1.7.3.2 and OpenSSL version 1.1.0g. Socat is a command line utility[4]that establishes two bidirectional byte streams and transfers data between them. The use of socat can be applied

[4] `http://www.dest-unreach.org/socat`

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 MultiTLS script starts by analyzing the arguments provided by the user. Afterwards, these arguments are used to execute socat commands. 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 executes 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 MultiTLS command arguments with the same names. 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 its name, 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. To create more tunnels, the IP of the last TUN interface created on the server side must be specified in the $ipServer argument.

TUN interfaces allow MultiTLS to create multiple virtual network interfaces. It is through the TUN interfaces that MultiTLS encapsulates the various tunnels. These interfaces operate at level 3 of the OSI model, and these devices can be used to establish VPN communications, as they allow the responsible software to encrypt the information before it is sent. MultiTLS uses several TUN interfaces, as each interface will allow establishing a TLS tunnel that will be encapsulated by the TLS tunnel of the next TUN interface. On the other hand, MultiTLS uses OpenSSL as a dependency, which allows performing all the cryptographic part, from creating and signing client and server certificates to the development of message ciphers. Whereas, the Socat dependency allows MultiTLS to establish multiple tunnels. This tool allows data transfer between two independent channels, being responsible for creating the TUN interfaces and using OpenSSL. That is, it is through Socat that the tunnel is established between the TUN interface on the client side and the TUN interface on the server side, using the implementation of OpenSSL in order to protect the connection.

3.7. Configurations

To successfully establish communication through the MultiTLS tool, it is necessary to ensure some configurations in the machines. In a first phase, the MultiTLS client uses port 4040 to send information that will be used to establish encrypted communication. This information includes: the IP address; the number of tunnels to consider in the MultiTLS communication to be established; the port on which this communication will be made; and the client certificate used in the first tunnel. On the server side, the information is received on port 4040 and sends its certificate to the first tunnel. The client receives the server's certificate on port 4040. Due to these initial negotiations, it is necessary to configure any firewalls that may interfere with the communication, to accept inbound and outbound data flow to port 4040. Once the initial negotiations are finished, the tool can now establish

k tunnels (defined by the user and less than or equal to four). The first tunnel is established through the port indicated by the user when starting the client-side program. For the remaining tunnels, the port number used will be incremented from the port initially indicated by the user.

If it is necessary to communicate between different networks, it is necessary to configure *port forwarding* to traverse a network gateway, such as a router. After this configuration, MultiTLS will operate transparently.

4. Evaluation

The experimental evaluation aims to answer questions about the performance and cost of MultiTLS. We have the following experiment sets: performance, file transfers, comparison with other approaches, and a use case.

4.1. Setup

For all the experiments, two virtual machines were used, one as a server and the other as a client, running on separate physical hosts.

Each presented measurement was repeated 100 times, with the computed average presented as the result. We assume a normal distribution, treating each run as a sample.

4.2. Performance

In this Section we assess the performance of MultiTLS. We want to answer two specific questions: *What is the cost of adding more tunnels? What is the cost of encrypting messages?*

In the following experiments, each virtual machine had 2 VCPUs, 8 GB RAM, and Ubuntu Linux 16.

In the first experiment, 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, the experiment consisted of using the iperf3 tool to measure 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 seen in Figure 3. We start with $k = 1$ so as to have as baseline a single encryption, i.e., we are not comparing against a scenario without security.

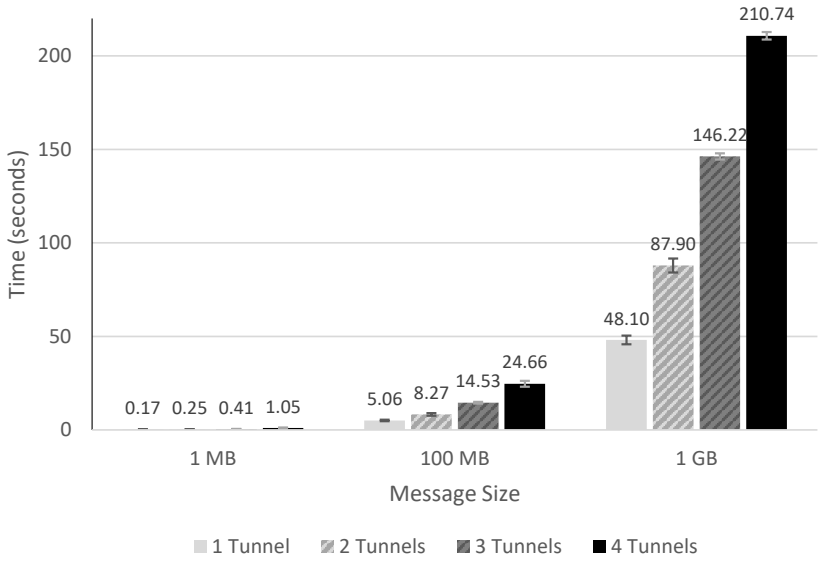

Figure 3: Comparison between the time it takes to send 1 MB, 100 MB and 1 GB messages in relation to the number of tunnels created.

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 $k = 1$.

We can answer the first question that for $k = 2$ the performance of MultiTLS is acceptable, since the time of sending messages with $k = 2$ is less than the double of 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

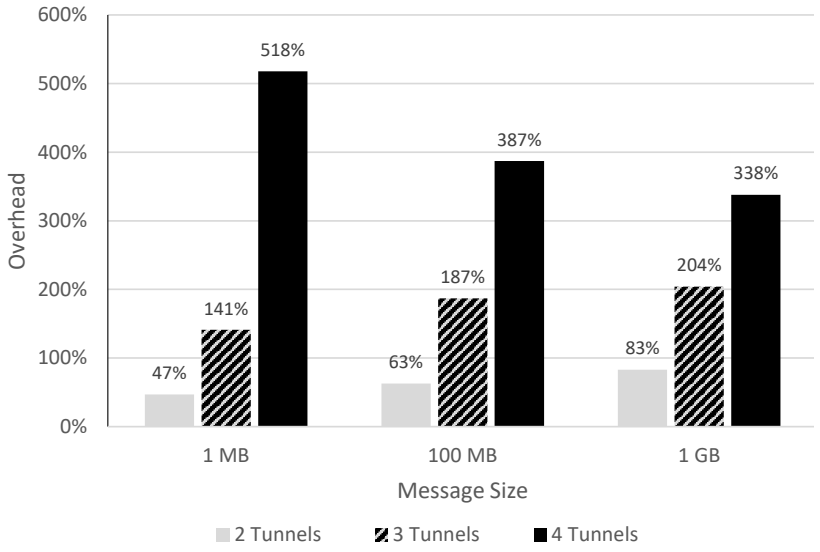

Figure 4: The overhead of adding more tunnels in relation to k = 1.

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 use of tunneling with multiple encapsulation layers can significantly impact network performance, a phenomenon known as "TCP meltdown" or "TCP-over-TCP collapse". TCP congestion control algorithms struggle to handle the complex feedback loops from multiple layers of tunneling, resulting in higher latency and degraded throughput (Harkanson et al., 2019).

The second experiment aims to evaluate the cost of encrypting the communication messages. To do this, using the same virtual 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. 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

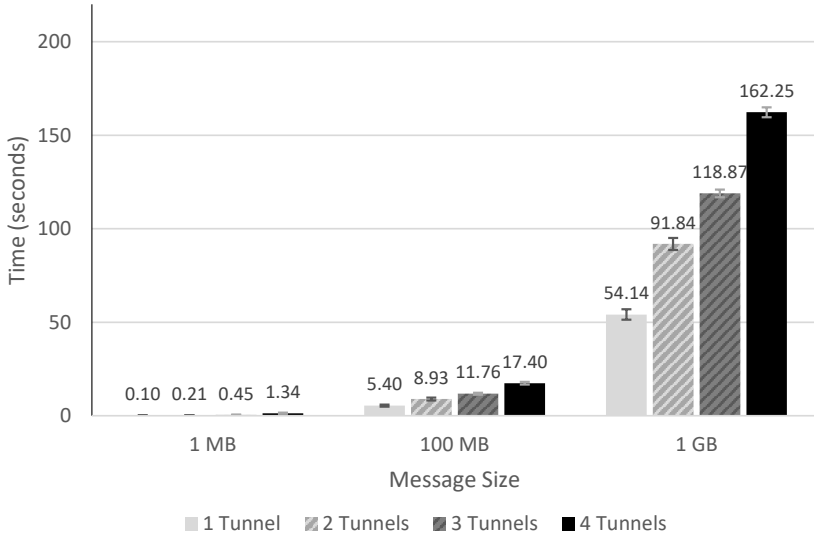

Figure 5: Comparison between the time it takes to send 1 MB, 100 MB and 1 GB messages in relation to the number of unencrypted tunnels.

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 6 shows the percentual difference between the first and second experiment, for each message size and k. We can see that, for certain message sizes and k, messages sent on the first experiment took less time than messages sent without encryption. 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: the time to encrypt the messages has a considerable low impact given that it takes 13% more time.

4.3. File transfers

For this set of experiments, we used machines 4 VCPUs, still 8 GB RAM, and Ubuntu Linux 20.04 LTS.

This scenario is based on the machine-to-machine for secure communication between legacy applications, described in Section 3.1. More specifically, FTP (*File Transfer Protocol*) was used. Through its client/server architecture, FTP is able to establish a connection between two points, which can be used to transfer files and perform other operations. The tests carried out

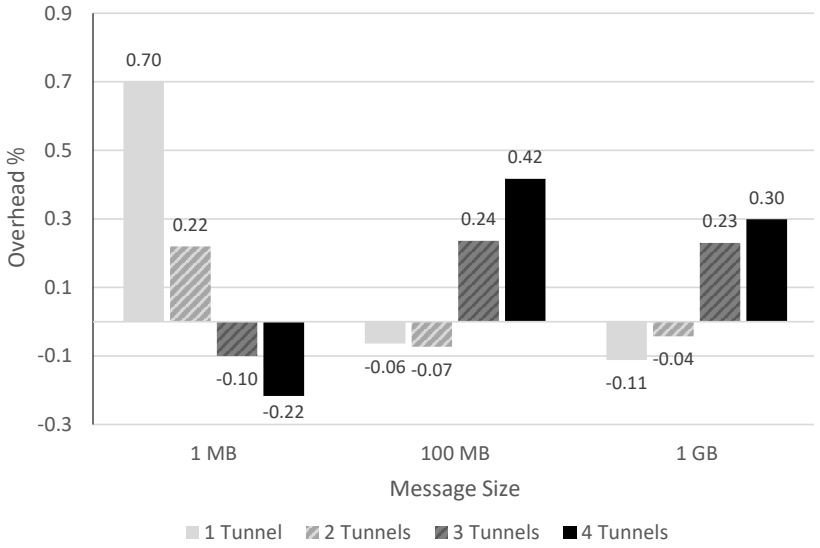

Figure 6: Difference between first and second evaluation results.

consisted of transferring different files (25 MB, 50 MB, 75 MB and 100 MB) and measuring the transfer time for different numbers of tunnels. Initially, as a reference, a test was performed for k = 0, that is, the measurements were made without using any tunnel, just a normal FTP communication. Then the same procedure was performed for k = 1, k = 2, k = 3 and k = 4, where k represents the number of tunnels used by the tool. In order to minimize the impact of possible disturbances on the network, all the results obtained were collected at the same time of day, under similar conditions. Another aspect that was taken into account was the representativeness of the data. In order to guarantee that the collected data sample was representative, the arithmetic mean and standard deviation of the obtained data was calculated. After collecting the samples, it was found that the standard deviation for all cases considered was less than one second.

The results obtained are show in tables related to the number of tunnels ($k = 0$, $k = 1$, $k = 2$, $k = 3$ and $k = 4$). Each table also has five columns, the first indicating the order in which the data were inserted, and the remaining four indicating the value of the measurements for the different files considered. Finally, each table also indicates the mean transfer time for each file.

The data referring to the transfer of files using FTP only ($k = 0$) is shown

in Table 1. Using only FTP ($k = 0$), it was possible to verify that the sending

k = 0				
	25MB File	50MB File	75MB File	100MB File
1st	0.08 s	0.20 s	0.24 s	0.31 s
2nd	0.08 s	0.18 s	0.24 s	0.33 s
3rd	0.07 s	0.15 s	0.24 s	0.26 s
4th	0.08 s	0.14 s	0.23 s	0.44 s
	...	...	...	...
Mean	0.08 s	0.17 s	0.22 s	0.33 s

Table 1: Results for file transfer with $k = 0$.

of files was practically instantaneous. Then, the same test was performed for $k = 1$, that is, using the tool with only one configured tunnel. The results obtained can be observed in Table 2.

k = 1				
	25MB File	50MB File	75MB File	100MB File
1st	2.90 s	3.78 s	6.93 s	9.86 s
2nd	2.25 s	3.63 s	6.63 s	9.81 s
3rd	3.25 s	2.86 s	7.50 s	11.90 s
4th	2.58 s	2.87 s	8.77 s	8.77 s
	...	...	...	...
Mean	2.87 s	3.30 s	7.39 s	9.54 s

Table 2: Results for file transfer with $k = 1$.

Through the observed data, it was possible to verify that when configuring a connection with only one tunnel, an increase in the average time is already noticeable. The remaining data, configured with two, three and four encapsulated tunnels, can be seen in the Tables 3, 4, and 5, respectively.

Figure 7 summarizes the data collected in the experiments, when using files of different sizes and different numbers of tunnels. The larger the file size and the number of tunnels used, the greater the transfer time. This result was expected, taking into account that not only was the size of the file itself increased, but also the *overhead* caused by the addition of tunnels to

k = 2				
	25MB File	50MB File	75MB File	100MB File
1st	3.00 s	8.38 s	14.41 s	19.16 s
2nd	4.14 s	7.28 s	16.09 s	20.38 s
3rd	2.07 s	7.63 s	13.92 s	21.38 s
4th	5.81 s	7.04 s	15.78 s	19.98 s
	...	...	...	...
Mean	3.41 s	7.70 s	14.46 s	20.26 s

Table 3: Test table for $k = 2$.

k = 3				
	25MB File	50MB File	75MB File	100MB File
1st	7.61 s	16.95 s	19.42 s	30.48 s
2nd	6.81 s	14.89 s	20.39 s	29.51 s
3rd	5.93 s	15.27 s	20.05 s	28.43 s
4th	5.48 s	15.54 s	19.51 s	30.76 s
	...	...	...	...
Mean	7.20 s	16.00 s	20.28 s	30.17 s

Table 4: Results for file transfer with $k = 3$.

k = 4				
	25MB File	50MB File	75MB File	100MB File
1st	10.95 s	19.91 s	26.68 s	39.35 s
2nd	10.44 s	18.54 s	27.11 s	38.83 s
3rd	9.07 s	19.66 s	26.31 s	40.08 s
4th	10.13 s	18.91 s	26.54 s	41.02 s
	...	...	...	... s
Mean	9.39 s	19.37 s	27.13 s	39.66 s

Table 5: Results for file transfer with $k = 4$.

the communication. The number of tunnels chosen for the communication has a more significant impact on the transfer time.

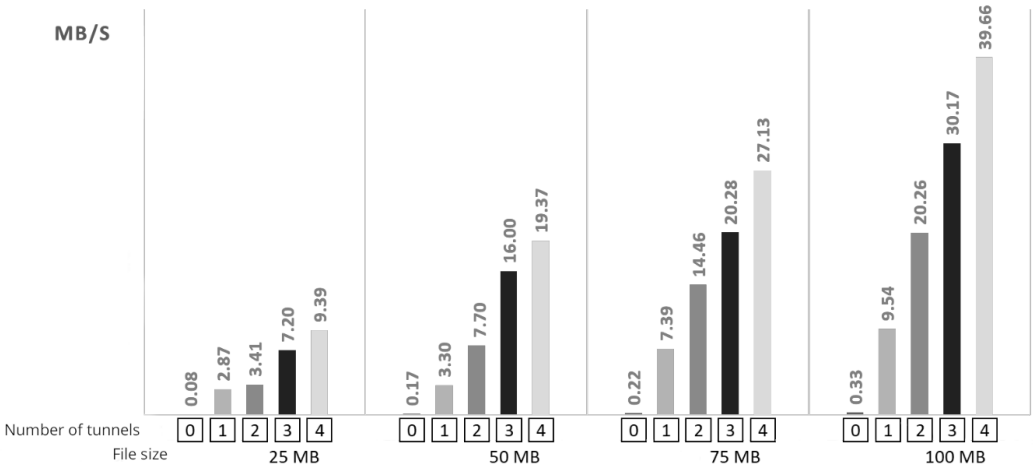

Figure 7: File transfer rate results (MB/s).

4.4. Comparison with vtTLS *and DTLS*

The purpose of this section is to compare the performance of MultiTLS with other tools and to know which of these approaches performs better. For this purpose, we use the same virtual machines as the experiment in Section 4.2. vtTLS is used to transfer three files each with the size of 1 MB, 100 MB and 1 GB. We ran vtTLS 100 times for each of these files. In addition to this experience, we also run a file transfer using a Datagram Transport Layer Security (DTLS) (Rescorla and Modadugu, 2012) 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 a channel that uses UDP over TCP, since with MultiTLS communication we have tunnels of several tunnels, that is, TCP over TCP. 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 8 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. Figure 8 also shows 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 because the server only sends the next fragment after receiving the size of the last fragment sent by it.

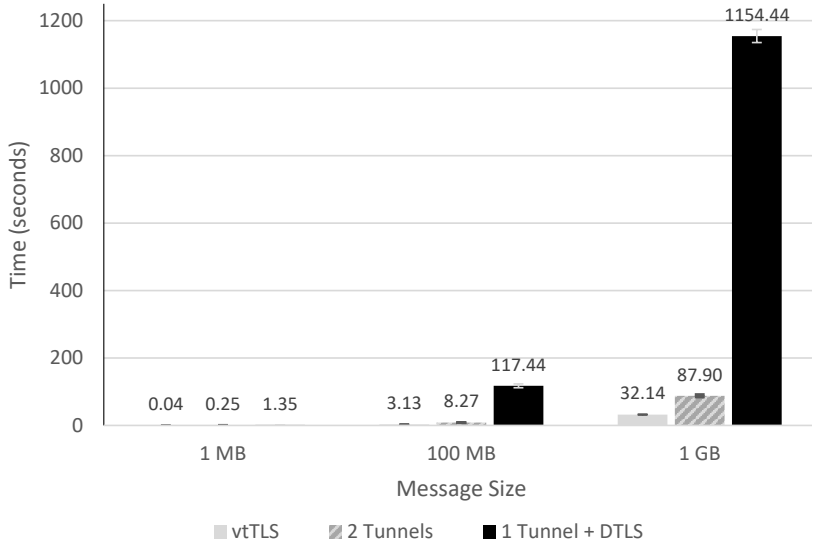

Figure 8: Time for sending messages with 1MB, 100MB and 1GB in size via vtTLS, 2 MultiTLS tunnels and 1 DTLS communication over 1 MultiTLS tunnel.

4.5. Browser to Web Proxy Performance

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 use case. We use MultiTLS to establish communication between a browser and a proxy, based on the scheme shown in Figure 2.

To do these experiments, one machine ran the Squid proxy, version 3.5.12, and the other ran the Google Chrome browser, version 66.0.3359.117.

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], 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. We collect the data with the browser development tools with the cache disabled.

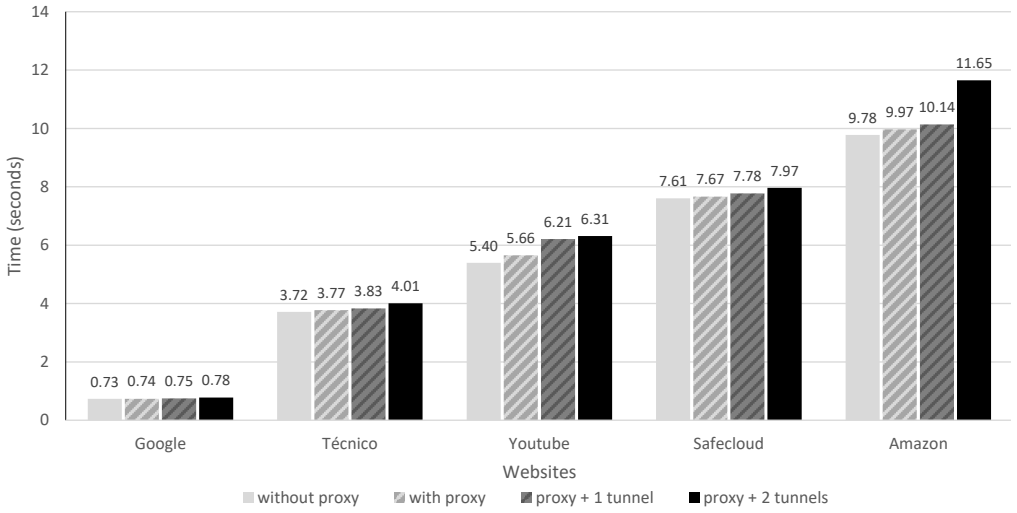

Figure 9: Time to load sites with: no proxy, with proxy, with proxy using MultiTLS with 1 tunnel and with 2 tunnels.

Figure 9 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. We can conclude that MultiTLS is a tool with good performance in tasks common to the day-to-day of many Internet users.

[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

5. Conclusion

We presented MultiTLS, a middleware that allows the creation of a channel of communication through the encapsulation of several secure tunnels in others. It increases security by using the diversity of cipher suites of the tunnels so that if $k-1$ cipher suites become insecure, there still remains cipher suite that protects the communication. MultiTLS has the advantage of not modifying any TLS implementation or any of its dependencies.

To evaluate MultiTLS, several tests were executed with the intention of measuring its performance and cost. The performance of file transfer was tested with different file sizes and different numbers of tunnels, confirming that these two variables have significant influence. The larger the file size, the greater the impact of the number of tunnels chosen on the transfer time. We also compared MultiTLS with the protocol vtTLS and we conclude that, although it performs less favorably in comparison, it has the advantage 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 even more secure, since the damage caused by vulnerabilities in one of these implementations does not endanger communication.

In our *future work*, we will focus on the following areas. Firstly, our research will concentrate on improving network tunnel performance, specifically addressing latency and bandwidth usage.

Next, we plan to port MultiTLS to other operating systems like Windows and Android/iOS to cater to an even broader range of use cases. Additionally, we will conduct testing on resource-constrained devices to validate the practical applicability of MultiTLS in securing Internet of Things applications. This presents a challenge due to the limitations of these devices, such as low-power processors, limited memory, and constrained communication protocols.

Another focus of our work is updating the diversity mechanisms for TLS version 1.3, that brings significant enhancements, including: resistance against downgrade attacks, simplified cipher suite negotiation, as well as support for the latest cryptographic algorithms.

Finally, we will build upon the groundwork laid by Carvalho (2014) on diversity measurements in cipher-suite selection. Our goal is to update the study and introduce diversity scoring for each cryptographic mechanism. This will deepen our understanding of diversity in cryptographic systems and pave the way for future solutions that will provide even greater security through diversity.

Acknowledgments

This work was supported by the European Commission project H2020-653884 (SafeCloud) and by Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID).

References

Biham, E., Shamir, A., 1993. Differential cryptanalysis of the data encryption standard.

Bilge, L., Dumitras, T., 2012. 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 , 833–844.

Bogdanov, A., Khovratovich, D., Rechberger, C., 2011. Biclique cryptanalysis of the full aes. Cryptology ePrint Archive, Paper 2011/449. URL: `https://eprint.iacr.org/2011/449`. `https://eprint.iacr.org/2011/449`.

Carvalho, M., Demott, J., Ford, R., Wheeler, D.A., 2014. Heartbleed 101. IEEE Security and Privacy 12, 63–67.

Carvalho, R.J., 2014. Authentication Security through Diversity and Redundancy for Cloud Computing. Ph.D. thesis. Instituto Superior Técnico, Universidade de Lisboa.

Estrin, D., Farinacci, D., Helmy, A., Thaler, D., Deering, S., 1995. IP in IP Tunneling. RFC 1853. Internet Engineering Task Force. URL: `https://tools.ietf.org/html/rfc1853`.

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

Harkanson, R., Kim, Y., Jo, J.Y., Pham, K., 2019. Effects of tcp transfer buffers and congestion avoidance algorithms on the end-to-end throughput of tcp-over-tcp tunnels, in: Latifi, S. (Ed.), 16th International Conference on Information Technology-New Generations (ITNG 2019), Springer International Publishing, Cham. pp. 401–408.

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

Kent, S., Seo, K., 2005. Security Architecture for the Internet Protocol. RFC 4301. Internet Engineering Task Force. URL: `https://tools.ietf.org/html/rfc4301`.

Khanvilkar, S., Khokhar, A.A., 2004. Virtual private networks: an overview with performance evaluation. IEEE Communications Magazine 42, 146–154.

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

Larson, R., Cockcroft, L., 2003. CCSP : Cisco Certified Security Professional Certification. McGraw-Hill/Osborne.

Littlewood, B., Strigini, L., 2004. Redundancy and Diversity in Security. Computer Security ESORICS 2004 , 227–246.

Menezes, A.J., van Oorschot, P.C., Vanstone, S.A., 1996. Introduction to public-key cryptography, in: Handbook of Applied Cryptography. CRC Press. chapter 8, pp. 355–422.

Nadeau, M., 2017. State of Cybercrime 2017: Security events decline, but not the impact.

NIST, 2001. Announcing the Advanced Encryption Standard (AES). Announcement. National Institute of Standards and Technology (NIST). URL: `https://csrc.nist.gov/publications/detail/fips/197/final`.

Postel, J., 1981. Internet Protocol. RFC 791. Internet Engineering Task Force. URL: https://tools.ietf.org/html/rfc791.

Rescorla, E., 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. RFC Editor.

Rescorla, E., Modadugu, N., 2012. Datagram Transport Layer Security Version 1.2. RFC 6347. RFC Editor.

Reynolds, J., Postel, J., 1994. Assigned Numbers. RFC 1700. Internet Engineering Task Force. URL: https://tools.ietf.org/html/rfc1700.

Rivest, R., 1992. The MD5 Message-Digest Algorithm (RFC 1321).

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

Rizzo, J., Duong, T., 2012. Crime: Compression ratio info-leak made easy, in: ekoparty Security Conference.

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

Shirey, R., 2007. Internet Security Glossary, Version 2. RFC 4949. RFC Editor.

Shor, P., 1996. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing 26, 1484–1509.

T. Ylonen, C. Lonvick, 2006. The Secure Shell (SSH) Protocol Architecture (RFC 4251).

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

Ylonen, T., Lonvick, C., 2006a. The Secure Shell (SSH) Authentication Protocol. RFC 4252. Internet Engineering Task Force. URL: https://tools.ietf.org/html/rfc4252.

Ylonen, T., Lonvick, C., 2006b. The Secure Shell (SSH) Connection Protocol. RFC 4254. Internet Engineering Task Force. URL: https://tools.ietf.org/html/rfc4254.

Ylonen, T., Lonvick, C., 2006c. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253. RFC Editor.