Diverse and Redundant Vulnerability-Tolerant Communication Channels

André de Matos Joaquim
andre.joaquim@tecnico.ulisboa.pt

Instituto Superior Técnico

Advisors: Professor Miguel Nuno Dias Alves Pupo Correia and
Professor Miguel Filipe Leitão Pardal

Abstract. Secure channels are an extremely important part of secure distributed systems by allowing two parties to exchange messages securely. An important example of standard protocol which provides such channels for application is SSL/TLS. However, there have always been concerns about the strength of some of encryption mechanisms used (e.g., DES) and some of them were regarded as insecure at some point in time (e.g., MD5). This project aims to present a solution to mitigate the problem of these channels being vulnerable to attacks due to vulnerabilities in its encryption mechanisms. In order to achieve this goal, our proposal consists in combining diverse and redundant cryptographic mechanisms to provide secure channels even if some of these mechanisms are vulnerable. Channels will rely on a combination of k mechanisms, with k being the diversity factor and $k \geq 1$. Although, combining cryptographic mechanisms is not trivial and may have the opposite effect. Our objective is to increase communication channels security. If one of the k mechanisms is insecure or has a vulnerability, the channel relies on another diverse redundant mechanism to maintain the channel secure. At most, $(k-1)$ mechanisms can be insecure with the channel remaining secure. The final intended outcome is a diverse secure communication channel which provides the same guarantees as SSL/TLS but is vulnerability-tolerant.

Keywords:
communicaton channels, diversity, redundancy, TLS, vulnerability-tolerance

Table of Contents

1 Introduction

Secure communication channels are mechanisms that allow two entities to exchange messages or information securely in the Internet. A secure communication channel has three properties: *authenticity*, *confidentiality*, and *integrity*. Regarding authenticity, in an authentic channel, the messages can not be tampered. Regarding confidentiality, in a confidential channel, only the original receiver of a message is able to read that message. Regarding integrity, no one can impersonate another. The information regarding the original sender of a message can not be changed.

Several secure communication channels exist nowadays, such as TLS, IPsec or SSH. Each of these channels is used for a different purpose, but with the same finality of securing the communication.

Transport Layer Security (TLS) is a secure communication channel widely used. Originally called Secure Sockets Layer (SSL), its first released version was SSL 2.0, released in 1995. SSL 3.0 was released in 1996, bringing improvements to its predecessor such as allowing forward secrecy and supporting SHA-1. Defined in 1999, TLS did not introduce major changes to SSL. Although, the introduced changes were enough to make TLS 1.0 incompatible with SSL 3.0. In order to grant compatibility, a TLS 1.0 connection can be downgraded to SSL 3.0, bringing security issues. TLS 1.1 and TLS 1.2 are upgrades which brought some improvements such as mitigating CBC (cipher block chaining) attacks and supporting more block cipher modes of operation to use with AES. TLS is divided in two sub-protocols, Handshake and Record, constituted by several mechanisms each. The Handshake protocol is used to establish or re-establish a communication between a server and client. The Record protocol is used to process the sent and received messages.

Internet Protocol Security (IPsec) is an Internet layer protocol that protects the communication at a lower level than SSL/TLS, which operates at the Application layer [22].

Secure Shell (SSH) is an Application layer protocol, such as SSL/TLS. SSH is a protocol used for secure remote login and other secure network services over an insecure network [53].

A secure communication channel becomes insecure when a vulnerability is discovered. Vulnerabilities may concern the protocol's specification, cryptographic mechanisms used by the protocol or specific implementations of the protocol. Many vulnerabilities have been discovered in SSL/TLS originating new versions of the protocol with renewed security aspects such as deprecating cryptographic mechanisms or enforcing security measures. Certain implementations of SSL/TLS have also been considered vulnerable by having implementation details causing a breach in security and affecting devices worldwide.

Our solution is a secure communication channel tolerant to vulnerabilities which does not rely on only one cipher suite. It is our belief that, in order to solve the problem, *diversity* and *redundancy* are helpful to mitigate vulnerabilities. Diversity and redundancy consist in using two or more different mechanisms with the same objective. For example, MD5 and SHA-3 are both hash functions

used to generate digests. In a real case, where MD5 has become insecure, our diverse secure communication channel resorts to other mechanism, SHA-3, in order to keep the communication secure. Using diversity and redundancy, when a mechanism is targeted by an attack, another mechanism is able to maintain the security and availability of the communication.

The rest of the document is organized in the following way: Section 3 presents the related work. Section 4 presents the architecture of the proposed solution. Section 5 presents how the solution is going to be evaluated. Section 6 presents the scheduling of the future work. Section 7 presents some conclusions.

2 Objectives

The objective of this project is to mitigate the problem of secure communication channels being vulnerable to attacks. The project involves studying the use of diversity and redundancy to improve security, in general, and to improve secure communication channels' security, in particular. More specifically, our proposal to achieve this objective is to develop a diverse secure communication channel tolerant to vulnerabilities that provides authenticity, confidentiality and integrity, and uses diverse and redundant mechanisms aimed at making the communication more secure. Our approach will introduce redundancy and diversity through specific entry points of our secure communication channel.

One of the challenges of this project is to study if diversity and redundancy have a real impact on increasing security of a communication channel while having reasonable performance and time-related costs which requires a precise evaluation of our solution.

3 Related Work

Before defining a solution, research was made in order to find previous studies about diversity, how encryption mechanisms can be combined and recent known attacks to secure communication channels, namely SSL/TLS. This section describes research done in these areas. Section 3.1 describes the SSL/TLS protocol and discusses some vulnerabilities of SSL/TLS. Section 3.2 refers to arguments presented by several researchers who defend the use of diversity in security and reviews several cases of diversity applied to software in order to make it secure. Section 3.3 discusses vulnerabilities in the most used cryptographic mechanisms and the changes made in order to correct those vulnerabilities. Section 3.4 shows how combining encryption mechanisms can be useful in order to increase the security of the final encryption mechanism.

3.1 SSL/TLS

A secure communication channel is a secure channel used by two entities to communicate with each other. A secure communication channel grants confidentiality, authenticity and integrity. Confidentiality is defined as keeping information – messages – secret. When Alice sends a message to Bob, only Bob can read that message. Confidentiality ensures that no one else will be able to read the message. Authenticity is defined as keeping information authentic. Authenticity ensures that no one tampered the message. Integrity is defined as keeping information honest. Integrity ensures that no one sends a message pretending to be someone else.

3.1.1 Brief history Secure Sockets Layer (SSL) [18] is a security protocol developed by Netscape Communications in 1994. SSL was created with the purpose of ensuring the existence of an open security standard and to provide privacy and reliability between two communicating applications. The first version of SSL, SSL 1.0, was never publicly released due to its several security flaws as it implemented weak cryptographic algorithms.

SSL 2.0, an improved version of SSL 1.0, was released in 1995. SSL 2.0 used RSA [39] for authentication and key exchange, and MD5 [38] for hashing. Several ciphering mechanisms could be used, such as Triple DES, with an effective key of 112 bits. SSL 2.0 was deprecated in 2011 due to strong security flaws including allowing man-in-the-middle attacks and supporting MD5, according to the RFC 6176 [49].

The third version of SSL, SSL 3.0, resulted from a collaboration between Netscape Communications and Paul Kocher. It was released in 1996 and its specification is regarded as a historic document for the Internet community. Among other improvements, SSL 3.0 allowed forward secrecy using Diffie-Hellman key exchange and supported SHA-1 for hashing. SSL 3.0 was deprecated in 2015 [3].

Transport Layer Security (TLS) [12] is the sucessor of SSL. The first version of TLS, TLS 1.0, defined in 1999, was an upgrade to SSL 3.0. Dramatic changes

were not made in this upgrade. However, those changes were enough to make TLS 1.0 incompatible with SSL 3.0. Although, TLS 1.0 supports a connection downgrade to SSL 3.0 which causes security issues. TLS 1.0 already supported all the authentication and key exchange mechanisms supported by the most recent version of TLS, TLS 1.2.

TLS 1.1 was released in 2006 as an upgrade from TLS 1.0. This version introduced a few changes in order to correct some flaws which allowed cipher block chaining (CBC) attacks.

TLS 1.2, released in 2008, is the third and current version of TLS. It includes several improvements over TLS 1.1 such as supporting more block cipher modes of operation to use with AES and supporting some variations of SHA-2 (see Table 3).

3.1.2 Explanation of the TLS protocol The TLS protocol has two main sub-protocols – the TLS Handshake Protocol and the TLS Record Protocol.

The TLS Handshake Protocol is a protocol used to establish or resume a secure session between two communicating parties – a client and a server. A session is established in several steps, each one corresponding to a different message (see also Fig. 1):

1. The client sends a *ClientHello* message to the server. This message is sent when the client wants to connect to a server or as a response to a *HelloRequest* from a server. The *ClientHello* message is composed by the client's TLS version, a structure denominated *Random*(which has a secure random number with 28 bytes and the current date and time), the session identifier, a list of the cryptographic mechanisms supported by the client and a list of compression methods supported by the client, both lists ordered by preference. The client may also request additional functionalities from the server.
2. The server responds with a *ServerHello* message. The *ServerHello* message is composed by the server's TLS version, a structure denominated *Random* analogous to *ClientHello*'s structure with the same name, the session identifier, a cryptographic mechanism and a compression method, both of them chosen by the server from the lists received in the *ClientHello* message, and a list of extensions.
3. If the key exchange algorithm agreed between client and server requires authentication, the server sends its certificate to the client – a *Certificate* message.
4. A *ServerKeyExchange* message is sent to the client after the *Certificate* message if the server's certificate does not contain enough information in order to allow the client to share a premaster secret.
5. The server then sends a request for the client's certificate – *CertificateRequest*. This message is composed by a list of types of certificate the client may send, a list of the server's supported signature algorithms and a list of certificate authorities accepted by the server.

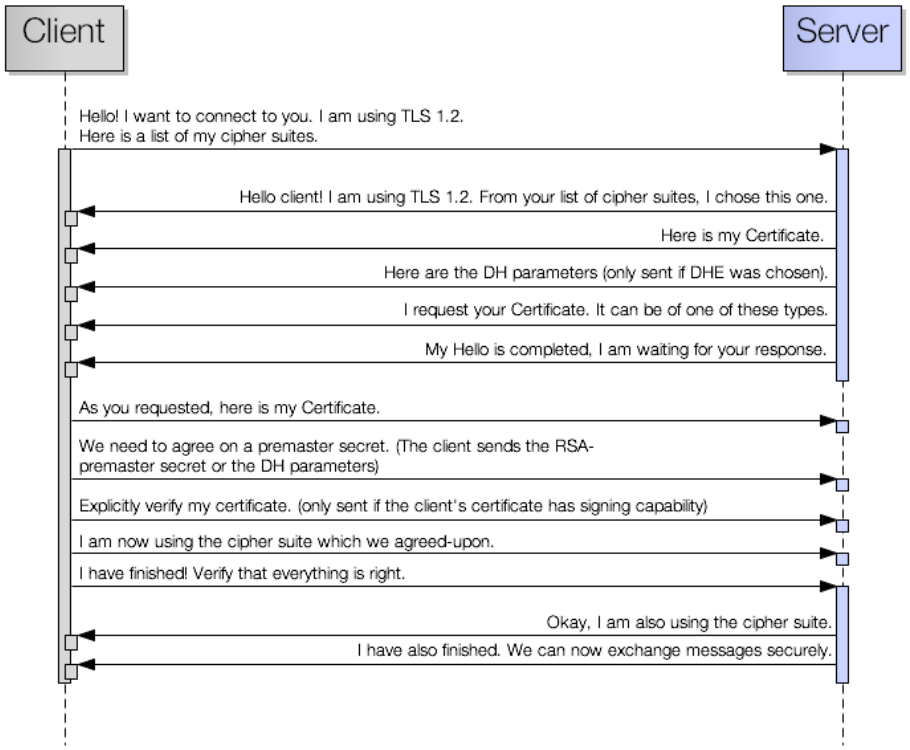

Fig. 1: Sequence diagram of the TLS Handshake Protocol, adapted from Section 7.3 of [12].

6. The server sends a *ServerHelloDone* message to conclude his first sequence of messages.
7. After receiving the *ServerHelloDone* message, the client sends its certificate to the server, if requested. When the server receives the client's certificate, there are some aspects that may risk the handshake. If the client does not send its certificate or if its certificate does not meet the server's conditions, the server may choose to continue or to abort the handshake.
8. The client proceeds to send to the server a *ClientKeyExchange* message containing an encrypted premaster secret. The client generates a premaster secret with 48 bytes and encrypts it with the server's certificate public key. At this point, the server and the client use the premaster secret to generate the session keys and the master secret.
9. If the client's certificate possesses signing capability, a *CertificateVerify* message is sent to the server. Its purpose is to explicitly verify the client's certificate.
10. The client must send a *ChangeCipherSpec* message to the server. A *ChangeCipherSpec* message is used to inform the server that the client is now using the agreed-upon algorithms for encryption and hashing. TLS has a specific

protocol to signal transitions in ciphering strategies denominated Change Cipher Spec. According to this protocol, a *ChangeCipherSpec* message must be sent by both client and server to notify one another that they are now using the algorithms agreed-upon.

11. The client now sends his last handshake message – *Finished*. The Finished message verifies the success of the key exchange and the authentication.
12. After receiving the *ChangeCipherSpec* message, the server starts using the algorithms established previously and sends a *ChangeCipherSpec* message to the client as well.
13. The server puts an end to the handshake protocol by sending a *Finished* message to the client.

By this point, the session is established. Client and server can now exchange information (application data) through the secure communication channel.

The TLS Record protocol is the sub-protocol which processes the messages to be sent and the received ones.

Regarding an outgoing message, the first operation performed by the TLS Record protocol is fragmentation. The message is divided into blocks called TLSPlaintext. Each block contains the protocol version, the content type, the fragment of application data and this fragment's length in bytes. The fragment's length must be 2^{14} bytes or less.

After fragmenting the message, each block may be optionally compressed, using the compression method defined in the Handshake. The compression operation transforms each TLSPlaintext block into a TLSCompressed block. Each TLSCompressed block also contains the protocol version and content type as TLSPlaintext block, and also contains the compressed form of the fragment and its length. The fragment's length must be $2^{14}+1024$ bytes or less. Although there is always an active compression algorithm, the default algorithm is CompressionMethod.null. CompressionMethod.null is an operation that does nothing, i.e., no fields are altered. In this case, the TLSCompressed blocks are identical to the TLSPlaintext blocks and no changes are made.

Each TLSCompressed block is now transformed into a TLSCiphertext block by encryption and MAC functions. Each TLSCiphertext block contains the protocol version, content type and the encrypted form of the compressed fragment of application data, with the MAC, and the fragment's length. The fragment's length must be $2^{14} + 2048$ bytes or less. When using block ciphers, it is also added a padding and its length to the block. The padding is added in order to force the length of the fragment to be a multiple of the block cipher's block length. When using AEAD ciphers, no MAC key is used.

The message is now sent to its destination.

Regarding an incoming message, the process is the inverse of the process for outgoing messages. The message is, therefore, decrypted, verified, decompressed, reassembled and delivered to the application.

3.1.3 TLS configuration variants As mentioned before, TLS does not oblige the use of pre-established algorithms. On the contrary, TLS supports several optional algorithms. The programmer implementing the protocol is given freedom to choose among those supported algorithms. TLS needs to specify, at least, a public-key algorithm, such as RSA, a symmetric key algorithm, such as AES, and a hash function, such as SHA-1. Table 1 presents all the key exchange algorithms supported by TLS [12]. Table 2 presents all the symmetric key algorithms supported by TLS. Table 3 presents all the hash functions supported by TLS.

A keyed-hash message authentication code (HMAC) is an algorithm that combines a secret key with an hash function to compute a message authentication code (MAC) to verify data integrity and authenticity of a message. Additional to the algorithms mentioned in Table 3, there is also a mechanism called authenticated encryption with associated data (AEAD) [41]. AEAD provides confidentiality for a ciphered message and, additionally, integrity and authenticity. AEAD provides a way of authenticating additional information attached to that message. The most recent member of the SHA family, SHA-3 (Keccak), is not yet supported by TLS 1.2.

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

Table 1: Key exchange algorithms supported by TLS 1.2. If the algorithm provides forward secrecy, it is marked with a (✓) in the third column. In case the algorithm is regarded as insecure, it is marked with a (✗) in the Secure column.

Cipher	RFC	Mode of operation	Type	Bits	Secure
AES	5288	GCM (Galois/Counter Mode)	Block	128, 256	✓
	6655	CCM (Counter with CBC-MAC)			✓
	-	CBC (Cipher block chaining)			✓
3DES EDE	-	CBC	Block	112	Weak
Camellia	6367	GCM	Block	128, 256	✓
	5932	CBC			✓
ARIA	6209	GCM	Block	128, 256	✓
	6209	CBC			✓
SEED	4162	CBC	Block	128	✓
RC4	-	———	Stream	128	✗

Table 2: Symmetric-key cipher algorithms supported by TLS 1.2. The Bits column refers to the strength of the algorithm. In case the algorithm is regarded as insecure, it is marked with a (✗) in the Secure column. 3DES EDE in not yet regarded as insecure but has low strength.

Algorithm	Hash function	Digest size in bits	Collisions found
HMAC-MD5	MD5	128	✓
HMAC-SHA1	SHA (SHA-1)	160	✗
HMAC-SHA256	SHA-256 (SHA-2)	256	✗
HMAC-SHA384	SHA-384 (SHA-2)	384	✗

Table 3: Algorithms supported by TLS 1.2 to check data integrity and authenticity.

3.1.4 TLS vulnerabilities TLS vulnerabilities can be classified in two types: specification vulnerabilities and implementation vulnerabilities. Specification vulnerabilities concern the protocol itself. A *specification vulnerability* regards a protocol's specification vulnerability which can only be fixed by a new protocol version or an extension. In this section, it is considered that the deprecation of a cryptographic mechanism is a protocol's design flaw as the protocol should not support an insecure mechanism. *Implementation vulnerabilities* are related to vulnerabilities in certain implementations of SSL/TLS, such as OpenSSL, or browers' implementations. The Internet Engineering Task Force (IETF) released RFC 7457 [45] on February 2015 containing the summary of known attacks to TLS specification and TLS implementations. In this section are presented some of the most recent attacks.

Starting with *specification vulnerabilities*, the most recent attack is called Logjam and it was discovered in May 2015 [1]. The Logjam attack consists in exploiting several Diffie-Hellman key exchange weaknesses.

Logjam is a man-in-the-middle attack that downgrades the connection to a weakened Diffie-Hellman mode. TLS supporting Diffie-Hellman with weak parameters is one aspect that makes this attack possible. The other aspect concerns about export cryptography. In the 90's, the United States of America legislated

some restrictions over exporting cryptography. In order to support SSL/TLS in some countries, not allowed to import U.S.A. cryptography, SSL/TLS supports weakened Diffie-Hellman modes – EXPORT modes [50]. Previous attacks, such as the one made possible due to the FREAK vulnerability [4], have already made use of this weakness. Adrian *et al.* consider Logjam a result of a protocol specification vulnerability due to the fact TLS still allowing the use of Diffie-Hellman with weak parameters [1].

In order to understand the attack itself, the concept of number field sieve algorithm has to be taken into account. A number field sieve algorithm is an efficient algorithm used to factor integers bigger than one hundred digits. The authors used a number field sieve algorithm to precompute two weak 512-bit Diffie-Hellman groups used by more than 92% of the vulnerable servers [1]. This approach was taken due to the fact that it is computationally heavy to generate prime numbers with certain characteristics. It is a safe measure for prime numbers with a safe length.

This man-in-the-middle attack exchanges the current cipher suites to DHE_EXPORT, forcing the use of weakened Diffie-Hellman key exchange parameters. As the server supports DHE_EXPORT, a completely valid Diffie-Hellman mode, the handshake proceeds without the server noticing the attack. The server proceeds to compute its premaster secret using weakened Diffie-Hellman parameters. From the client's point-of-view, the server chose a seemingly normal ephemeral Diffie-Hellman (DHE) option and proceeds to compute its secret also with weak Diffie-Hellman parameters. By this point, the man-in-the-middle can use the precomputation results to break one of the secrets and establish the connection to the client pretending to be the server. One aspect worth noticing is that this attack will only succeed if the server does not refuse to accept DHE_EXPORT mode.

The solution for this vulnerability is simple and has already been implemented. Browsers simply deny the access to servers using weak Diffie-Hellman cipher suites, such as DHE_EXPORT, although TLS still allows it.

Another attack concerning the protocol specification is a padding attack named POODLE [34]. POODLE stands for Padding Oracle On Downgraded Legacy Encryption and it was discovered by Google engineers in 2014. The origin of this attack is the backwards compatibility for SSL 3.0 by many TLS implementations. To be successful, the attacker should induce a downgrade attack first, in order to transition from TLS 1.x to SSL 3.0.

POODLE targets the CBC mode of operation used in SSL 3.0. Although, an assumption that the attacker can modify communications between the client and the server must be made. POODLE is a padding attack which relies on that fact. As the CBC padding is not deterministic and not covered by the MAC, the integrity of the CBC padding is not fully verified when decrypting.

The POODLE attack can be used to decrypt HTTP cookies in web sites. The authors state that for every byte revealed, 256 SSL 3.0 requests are needed. The attack has been proved to be possible in TLS [25]. The implementations affected are those that do not properly check the padding used.

The most obvious solution is to avoid SSL 3.0 by disallowing the backwards compatibility and deprecating SSL 3.0. Another solution is to use TLS Fallback SCSV, as described in [33]. This signaling cipher suite value (SCSV) intends to prevent unnecessary downgrade of the connection when both client and server actually do support the most recent version of TLS.

Regarding implementation vulnerabilities, Heartbleed, discovered in 2014, is one of the most recent ones. Its name is originated from the mechanism where the vulnerability lies, the heartbeat extension. Heartbleed was a security vulnerability in OpenSSL 1.0.1, through 1.0.1f, when the heartbeat extension [44] was introduced and enabled by default. The Heartbleed vulnerability allowed an attacker to perform a buffer over-read [8]. A buffer over-read happens when is read more data than allowed.

Heartbleed can be explained in a very simple manner. OpenSSL trusted the client's input correctness. In a normal heartbeat request, the client sends a message to the server saying: "If you are alive, send me "Bob" which has 3 letters". The server responds "Bob". In a abnormal heartbeat request, the client sends a message to the server saying "If you are alive, send me "Bob" which has 500 letters". The server responds "Bob" followed by 497 bytes taken from the positions of memory following where "Bob" was stored.

The server did not check if the real length of the word matched the length the user said the word had.

3.2 Implementing diversity in security

There are two main decisions which a programmer has to make while implementing diversity – when does he introduce the diversity and what should he diversify in his software [26]. Software has a rather complex life cycle that can last some time. There are many opportunities to introduce diversity – earlier, in the implementation phase, later, when patches and updates are applied, or somewhere in the middle. This is a key design decision because it affects the performance and the level of security. This will determine also if the diversification is applied to binary or source code.

For example, for programs written in C or C++ , the code is compiled, assembled and then linked to produce machine-code readable by the processor. If the objective is to commercialize the program, it is then packaged for distribution and installation on the users' computer. Different approaches in implementing diversity use different points of the program life cycle as the entry point for diversity, and introduce different diversity in distinct ways.

In practice, a specification of a program can be implemented in several different ways. Each programmer has his coding style and his favourite libraries, algorithms or design patterns. Diversity exploits this property by randomizing program implementations in order to dismantle attacks. Concluding, knowing what to diversify and when to introduce that diversity are two main core aspects of implementing diversity in a program and together they determine the fundamental properties of any specific approach.

3.2.1 Arguments for diversity in security Nowadays, one of the main security concerns is a system being static. A static system is a system where there are no changes with time. It is clear that a static system, or sitting-target, is more prone to attacks than a diverse system because, in static systems, an attacker has time to study the system's architecture and vulnerabilities. This way, an attacker can perform several distinct attacks while the system remains the same.

On the opposite side, a diverse system is a system that changes with time. The term *diversity* describes multi-version software in which redundant versions are deliberately made different from one another [27]. Without diversity, there is just one program spread across devices around the world. Every device has the same program version, with the same vulnerabilities. Using diversity, we can present the attacker with modified programs, each one with modifications. There is no proper way of knowing the attacks which that particular program is susceptible to.

Software diversity targets mostly implementation details and the ability of the attacker to replicate the user's environment. Diversity does not change a program's logic, just its implementation details. So, it will not be useful if a program is badly implemented.

3.2.1.1 Moving-target defenses Moving-target defense aims to overcome the problems caused by static defense mechanisms by dynamically altering properties of programs and systems [15].

In daily life, people use some moving-target defenses without even noticing. Periodically changing passwords or IP addresses are just some examples of actions which introduce some diversity in the system and protect against some common attacks.

Static systems allow the attackers to study the system for a long time as it is not diversified with time. For example, if the password is not changed, an attacker can discover it using, per example, a brute force attack, in a limited amount of time.

After attackers reconnoitre the system, they can build their attacks and all the defenders can do is build monitors and shields to defend against those attacks [9]. This brings a lot of costs to the defenders, who have to protect themselves against all types of possible attacks. Concerning the attacker, as long as the system remains the same, all he has to do is to try different attacks and see which one is effective.

With moving-target defenses, the attacker has limited time to understand the system and to find vulnerabilities on that system. With that being said, some attacks performed by an attacker might become useless once the system performs dynamic changes to itself.

There are two types of moving-target defenses: reactive and proactive [9]. Proactive defenses are generally slower than reactive defenses as they prevent attacks by increasing the system complexity periodically. Reactive defenses are faster than proactive defenses as they are activated when they receive a trigger from the system when an attack is detected. This may cause a problem where an

attack is performed but it is not detected. In this case, the reactive defenses are worthless, but the proactive defenses can prevent that attack from happening.

Proactive defenses must be more sparse in time but they are efficient because they can anticipate a potential attack. At the same time, reactive defenses are useful to prevent attacks that might surge and that were not anticipated by the proactive defenses. The best approach is to implement both – adaptive response. It is needless to say that these defenses are only as good as its ability to make an very unpredictable change to the system.

Although moving-target defense is a good defensive measure, it also has negative aspects. Moving-target defenses increase the system's complexity and it is not one hundred per cent secure as attackers can also react to the system changes with changes in the attack and still be successful. Also, the moving-target defenses change unpredictably the system not only for the attacker but also for the defenders. This can make the system difficult to maintain and to track possible issues in it.

3.2.2 N-version programming The concept of *N-version programming*, or multi-version programming, was first introduced by Chen and Avižienis in 1977 [2]. N-version programming is defined as the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification – *versions*.

Although redundancy is useful in order to enable a program to recover from faults, it does not improve the reliability of a program. Given the same input and state, the program will produce the same error. Chen and Avižienis also state that if we want to use redundant software to achieve software fault-tolerance and, consequently, reliability, then it should meet three constraints:

1. It does not require complete self-checking;
2. It does not rely on run-time software diagnosis;
3. It must contain independently developed alternatives routines for the same functions.

It is the last constraint that concerns us – a need arose to have diversity in redundant software.

The N from N-version programming comes from the N diverse versions of the program by N different programming teams, or programmers, that do not interact with each other regarding the programming process.

Although N-version programming seems a very good form of diversity, it has some issues. First of all, even though the separate individuals or groups that program different versions of the program do not interact with each other, there is no guarantee that a certain vulnerability is not going to be in all of those N-versions of the program. It might be an algorithm vulnerability or just a programming bug or flaw shared by all versions. Other limitation of the N-version programming is that every version is originated from the same initial specification. With that being said, we need to assure the initial specification's correctness, completeness and unambiguity prior to the versions development.

3.2.3 Profile-guided automated software diversity Code injection is a type of attack that consists in injecting code into a vulnerable program. A major prevention technique to this kind of attack called W⊕X appeared in 2003 and was implemented in the OpenBSD 3.3 [11]. This technique prevents a page from being executable and writeable at the same time. When attackers realized they were not able to inject code anymore, they came up with the idea to reuse code from the program itself to perform the attack. In order to counter these attacks, Homescu *et al.* suggested an approach using software diversity and NOP insertion [21].

A NOP instruction is an instruction which execution has no effect on the processor or memory. Also, NOP instructions do not affect the logic of a program. The insertion of NOP instructions diversifies the code layout and presents a powerful counter to attacks that reuse tiny code fragments, named *gadgets*, such as Return-Oriented Programming (ROP), a technique by which an attacker can introduce arbitrary behaviour in a program whose control flow he has diverted, without injecting any code [40]. Although it seems to be a good technique, the insertion of NOPs can produce a high performance overhead, even if combined with profiled data.

3.3 Vulnerabilities in cryptographic mechanisms

This section presents some vulnerabilities in cryptographic mechanisms, specifically in some of the mechanisms supported by the TLS protocol (see Section 3.1.3). As stated in Section 3.1, TLS may use a variety of cryptographic mechanisms. Not every mechanism supported by TLS is secure. Several of these mechanism have vulnerabilities which can make the communication insecure, if exploited. In this section, we study the vulnerabilities of different cryptographic mechanisms supported by TLS. As our solution intends to use diversity and redundancy to increase security, studying the known vulnerabilities of these mechanisms will help us understand which are the most common vulnerabilities and which mechanisms are secure.

3.3.1 Vulnerabilities in asymmetric cipher mechanisms Proposed by Rivest *et al.* in 1978, RSA is a cryptographic mechanism used to cipher and sign messages or information. RSA's security is based on two problems: factorization of large integers and the RSA problem [30]. RSA's strength is inversely proportional to one's available computational power. As the years pass, it is expected that performing the factorization of large integers becomes a feasible task. RSA will be broken when those problems can be solved within a small amount of time.

Kleinjung *et al.* performed the factorization of RSA-768, a RSA number with 232 digits [23]. The researchers state they spent almost two years in the whole process, which is clearly a non feasible time. Factorizing a large integer is a very different concept from breaking RSA. RSA is still secure (see Table 1). As of 2010, the researchers concluded that RSA-1024 would be factored within five

years, i.e., in 2015. As for now, no factorization of RSA-1024 has been publicly announced, so their conclusion proved to be false.

Shor, using quantum computing, invented an algorithm to factorize integers in polynomial time, named *Shor's algorithm* [46]. One obstacle arises when we try to implement Shor's algorithm – it is yet to be invented real quantum computers. For the time being, quantum computers are just theoretical models.

Other attacks non related to the problems above mentioned can target RSA. A chosen plaintext attack [13] is an attack which exploits the fact that RSA is deterministic – given the same input, the same output will be produced. Let's assume the attacker has the public key, knows some of the content of the original message, and has the original message ciphered. He can try to cipher multiple plaintexts of his own with the public key, compare them with the original ciphered message, and discover the message.

A chosen ciphertext attack [36] is another possible attack to RSA. It benefits from the multiplicative property of RSA. The product of two ciphertexts is equal to ciphering the product of the two plaintexts. In these attacks, an attacker sends a ciphered message of his own, consisting on the original ciphered message M multiplied by a random value R, ciphered with the same key as M. If the attack is successful, the attacker can now obtain M by decomposing the message sent. These attacks assume that an attacker can obtain plaintexts from ciphertexts from the system. Adaptive chosen ciphertext attacks are an adaptive form of chosen ciphertext attacks in the sense that they use the results of previously sent ciphertexts to select future ciphertexts.

As shown, the attacks do not rely in breaking the essence of RSA, mathematical complexity, but rather focus on other details simpler to attack.

3.3.2 Vulnerabilities in symmetric cipher mechanisms Triple DES, or 3DES, is an encryption mechanism based on DES [16]. As the name says, Triple DES consists on multiple encryption using DES. Triple DES can be implemented using one of three keying options:

1. All three keys are independent;
2. Key 1 is equal to Key 3. Key 1 and Key 2 are independent;
3. All three keys are equal.

Associated with each keying option there is a set of vulnerabilities. Keying option one uses three different 56-bit keys, i.e., a 168-bit key, but due to meet-in-the-middle (MITM) attacks, the real security is 112-bit. Lucks presented an attack to the keying option one of Triple DES [28]. The proposed attack consists in a meet-in-the-middle attack which reduced the number of steps needed to roughly 2^{108} steps from 2^{112} steps. While being an improvement compared to the brute force attack, this attack still requires a lot of time to be performed, which makes it impracticable.

NIST states that Triple DES with keying option one is viable until the end of 2030. From there onwards, its use is disallowed [35].

Keying option two uses two different 56-bit keys, i.e., a 112-bit key. Keying option two, also known as two-key triple encryption, can be attacked using chosen plaintext attacks with about 2^k steps, $k = 56$ [31]. Merkle *et al.* concluded it is preferable to use a single encryption algorithm with a longer key rather than a multiple encryption algorithm with a smaller key. Keying option three has the same security as DES, which makes it an insecure option.

The Advanced Encryption Standard (AES), originally named Rijndael, is an encryption mechanism created by Rijmen and Daemen [37]. AES can be employed with different key sizes – 128, 192 or 256 bits. The number of rounds corresponding to each key size is, respectively, 10, 12 and 14. AES has become the standard encryption mechanism, used by many protocols such as TLS. The growth of its popularity attracted attackers to try to find vulnerabilities in it.

The most successful cryptanalysis of AES was published by Bogdanov *et al.* in 2011, using a biclique attack, a variant of the MITM attack [6]. It performed slightly better than brute force attacks. In order to understand how difficult it is to do the cryptanalysis of AES, this attack achieved a complexity of $2^{126.1}$ for the full AES with 128-bit (AES-128). The key is therefore reduced to 126-bit from the original 128-bit, but it would still take many years to successfully attack AES-128. Prior to Bogdanov *et al.*, when Rijndael was just a candidate to AES, Ferguson *et al.* presented, in 2010, the first known attacks on the first seven and eight rounds of Rijndael [17]. Although it shows some advance in breaking AES, AES with a key of 128 bits has 10 rounds. Related-key attacks were also used to try to exploit AES [5] but no properly designed software will use related-keys. Side-channel attacks are another type of attacks to AES. These attacks rely on certain unexpected outputs of the computational execution of AES, such as the power consumed, the heat generated or the time taken to perform the operation.

As of today, there are no known practical attacks that succeeded in breaking AES encrypted data.

3.3.3 Vulnerabilities in hash functions After mentioning the existent, or non-existent, vulnerabilities for ciphering mechanisms, this subsection refers to vulnerabilities in hash functions. A hash function is a function that outputs a hash given a string as input. Hash functions' main applications in modern cryptography regard data integrity and message authentication. Sometimes also called message digest or digital fingerprint, the hash is a compact representation of the input string and can be used to uniquely identify that hashed input string [29].

According to Menezes *et al.*, a hash function h must have a minimum of two properties:

- compression – h maps an input string s of finite length to an output $h(s)$ of a fixed length x.
- ease of computation – given h and an input string s, the hash $h(s)$ is easy to compute.

Menezes *et al.* also list three potential properties, additionally to those above:

- preimage resistance – for all pre-specified outputs, it is computationally infeasible to find any input string which hashes to that output, i.e., finding the input string s, given the output $h(s)$.
- 2nd-preimage resistance – it is computationally infeasible to find any second input string which has the same hash as any specified input string, i.e., given s_1, finding $s_2 \neq s_1$, where $h(s_1) = h(s_2)$.
- collision resistance – it is computationally infeasible to find two input strings whose hash is identical, i.e., finding $s_1 = s_2$, where $h(s_1) = h(s_2)$.

If a hash function is not preimage resistant or 2nd-preimage resistant, it is therefore vulnerable to preimage attacks. If a hash function is not collision resistant, it is vulnerable to collision attacks. Some generic attacks to hash function include brute force attacks, birthday attacks and side-channel attacks.

MD5, the succesor of MD4, was created by Rivest in 1991 [38]. MD5 is a cryptographic hash function that, although being proved to be insecure, is still widely used nowadays. MD5 produces a 128-bit message digest and is commonly used to verify data integrity. Wang *et al.* proved in the year of 2005 that MD5 is not collision resistant [52]. The employed attack was a differential attack which is a form of attack that consists in studying how differences in the input affect the output.

The Secure Hash Algorithm 1 (SHA-1) is another cryptographic hash function which produces a 160-bit message digest. Although there have not been publicly found actual collisions for SHA-1, it is considered insecure and it is recommended to use SHA-2 or SHA-3 [43]. Other attacks have been successful against SHA-1. Stevens *et al.* presented a freestart collision attack for SHA-1's internal compression function [48]. Taking into consideration the Damgard-Merkle [32] construction for hash functions, and the input of the compression function, a freestart collision attack is a collision attack where the attacker can choose the initial chaining value, also known as initialisation vector (IV). Although, freestart collision attacks being successful does not imply that SHA-1 is insecure, but it is a step forward in that direction.

In 2005, Wang *et al.* presented a collision attack on SHA-1 that reduced the number of calculations needed to find collisions from 2^{80} to 2^{69} [51]. The researchers claim that this was the first collision attack on the full 80-step SHA-1 with complexity inferior to the 2^{80} theoretical bound. By the year of 2011, Stevens improved the number of calculations needed to produce a collision from 2^{69} to a number between $2^{60.3}$ and $2^{65.3}$ [47].

Nowadays it is still computationally expensive to perform these number of calculations. In the near future, by the year of 2021, a collision attack is expected to be affordable to a university research project [42].

3.4 Combining encryption mechanisms

Our proposal includes diversity and redundancy of encryption mechanisms by combining two or more different mechanisms in several steps of the communication. Nevertheless, combining those mechanisms is not trivial.

Encryption is the process of encoding a message, or information, so that only the authorized personnel to read or access the message, or information, can actually read or access it. A cipher is an algorithm that is used to encode or decode messages. The sender uses a cipher algorithm in order to cipher the message. The receiver of that message must also use the same cipher algorithm to decipher the message and be able to read it. Encryption mechanisms can be categorized by two parameters: the type of key or the type of input data. Encryption mechanisms can use two types of keys: asymmetric or symmetric.

In *asymmetric encryption*, also known as public-key encryption, each user has a pair of keys, a private key and a public key, of his own. If Alice wants to send a message to Bob, she ciphers the message with Bob's public key. When Bob receives the message, he then uses his private key, which corresponds to the inverse of his public key, to decipher the message. This way, only Bob can read the message that Alice sent specifically to him. A known example of an asymmetric key algorithm is RSA. *Symmetric encryption* is associated with a key shared by both sender and receiver – secret key. Alice and Bob share a key which is used to both cipher and decipher messages. An example of a symmetric key algorithm is AES.

Both of these models have their strengths and weaknesses so several solutions were found to solve their issues. Asymmetric encryption mechanisms tends to be slower. On the opposite side, using symmetric encryption mechanism, both users need to come up with a way to exchange keys in a secure fashion, which can be a problem.

As for the type of input data, we can have one of two types: block ciphers or stream ciphers. Block ciphers are characterized by encrypting blocks of data of fixed size and stream ciphers are characterized by encrypting streams of data. Symmetric encryption can use block or stream cipher but asymmetric encryption can not use stream ciphers.

Several people tried to combine symmetric encryption with asymmetric encryption and different approaches surged with time.

3.4.1 Hybrid encryption The idea of combining symmetric encryption with asymmetric encryption arose. Asymmetric encryption has not a very good performance comparing to symmetric encryption and there is a problem in securely distributing the shared key needed for symmetric encryption.

Hybrid encryption uses both symmetric and asymmetric encryption. It consists in ciphering a shared key using the asymmetric encryption. Pretty Good Privacy (PGP) uses hybrid encryption and follows the OpenPGP standard (RFC 4880) [7] for the encryption and decryption of data. First of all, a random symmetric key is generated. This symmetric key is used by a symmetric encryption mechanism to cipher the message to be sent. This is done because ciphering with a symmetric encryption mechanism tends to be faster than with an asymmetric encryption mechanism, as stated above. In PGP, every symmetric key randomly generated is called session key and is used only once. PGP then uses an asymmetric encryption mechanism to cipher again the message (previously ciphered

with the symmetric encryption algorithm) using the receiver's session key. This way, only the receiver can decrypt the message with his private key.

According to the OpenPGP standard, several symmetric and asymmetric algorithms can be used. As for public-key algorithms, the most commonly implemented is RSA, with keys of size greater than or equal to 1024 bits. Concerning symmetric-key algorithms, RFC 4880 obliges to, at least, implement 3DES. Although, the current standard is AES-128.

Fujisaki *et al.* proposed in 2005, a new hybrid encryption mechanism [19] that is chosen ciphertext secure, following a set of assumptions:

- Asymmetric encryption has one-wayness property;
- Symmetric encryption has one-time security;
- Any plaintext of the asymmetric encryption has, at least, $2^{\omega(logk)}$ possible cipher texts.

Theoretically, this hybrid encryption scheme seems to be very good, the authors did not perform practical evaluation of the scheme. Therefore, there is no comparison being made with any other hybrid encryption scheme already existent.

Kurosawa-Desmedt [24] was another hybrid encryption scheme proposed by Kurosawa and Desmedt in 2004 using a key encapsulation mechanism (KEM) that was not necessarily chosen ciphertext secure. Further research on the Kurosawa-Desmedt proved the key encapsulation part is not by itself chosen ciphertext secure. This was proved by presenting a simple chosen ciphertext attack (CCA) attack on the Kurosawa-Desmedt's KEM [10].

3.4.2 Multiple encryption Multiple encryption consists in using more than one cipher mechanisms to cipher a message or information. Multiple encryption is used to strengthen the security over simple encryption. Multiple encryption can be employed using different or identical cipher mechanisms. Double encryption was one of the first multiple encryption mechanisms. DES was considered insecure due to its small 56-bit key [31]. The solution found was to encrypt twice, using two consecutive DES with two independent keys. Diffie and Hellman proved that this solution is susceptible to a known plaintext attack with 2^{56} operations [14]. Therefore, the suggested approach is to use triple DES encryption also known as TripleDES or 3DES. As mentioned before, this mechanism provides 112-bit real security due to being susceptible to a meet-in-the-middle attack which requires 2^{112} operations to succeed.

Multiple encryption, while seemingly improving security, is not always a good solution. In some cases, multiple encryption can have the reverse intended outcome by decreasing security. In symmetric encryption, the mechanism to encrypt is also the mechanism to decrypt. If one encrypts twice a message or information using symmetric encryption, the first layer of encryption will cipher the message and the second layer will undo the encryption made by the first layer. Multiple encryption is also less secure than a single encryption mechanism designed to use a longer key. Although, multiple encryption consists in aggregating several

encryption mechanisms to increase security, the aggregate may be susceptible to some other attacks, such as the double-DES, and, therefore, not achieving the goal.

3.4.2.1 Layered encryption Layered encryption can be considered a sub-type of multiple encryption. Layered encryption consists in ciphering layers of information or messages with identical or similar cryptographic mechanisms using different keys. One example of layered encryption is the encryption model of onion routing [20]. Onion routing provides real-time, bi-directional and anonymous communication for any protocol which can be modified to use a proxy. Onion routing's anonymity is based on the fact that an onion router only knows its immediate predecessor and its successor. Onion routing use a structure called onions. Onions are composed by layers of encrypted data structures, each layer corresponding to a routing node or "onion router". Each onion router peels its layer and sends the message forward to the next onion router.

4 Architecture

Our proposal is a diverse secure communication channel. It aims to increase security using diverse and redundant mechanisms and it is based on the TLS 1.2 standard.

Our proposal solves the problems originated by having only one cipher suite negotiated between client and server. In the case when one of the cipher suites mechanisms is insecure, the secure communication channels using that cipher suite may be vulnerable. Our diverse secure communication channel negotiates more than one cipher suite between client and server and, consequently, more than one encryption mechanism will be used for each purpose. Hence, the channel does not rely in only one cipher suite. Although most cipher suites used by TLS 1.2 are regarded as secure, as stated in Section 3.1.3, there is not any assurance that a governmental agency or a company with high computational power is not able to break that encryption mechanism today or in the near future.

Diversity and redundancy's primary entry point in our proposal is in the Handshake where client and server negotiate the k cipher suites to be used, where k, $k \geq 1$ is the diversity factor, representing the number of different cipher suites. In a case when the diversity factor is 1, it is considered that the channel has no diversity, and the channel becomes regular TLS 1.2.

Even when $(k-1)$ cipher suites become vulnerable, our proposal remains secure due to the diversity existent. The remaining redundant cipher suite ensures that the communication is secure by remaining invulnerable.

Nevertheless, not all ciphers suites are compatible with each other. As referred in Section 3.4.2, cipher suites must be combined in a way that security increases and not the opposite. The server chooses the best combination of k cipher suites according to the cipher suites the client has available.

Diversity and redundancy will also be introduced in the following communication between client and server. It is our intent to use a subset of the k cipher suites defined in the Handshake Protocol to cipher the messages. While performance must be taken into account, we will proceed to estimate the reasonable k cipher suites, and also the reasonable subset of k to be used to cipher the messages.

4.1 Handshake Protocol

Our proposal's Handshake Protocol is similar to TLS Handshake Protocol. Messages' name are identical which provides easier understanding, migration and transition from TLS. In this section, all the messages' names are analogous to TLS 1.2's messages. Using this simplification, anyone who is aware of TLS Handshake Protocol can easily understand our proposal's Handshake messages and their purpose.

The messages which are diversity and redundancy entry points are *ClientHello*, *ServerHello*, *ServerKeyExchange*, *Certificate*, and *ClientKeyExchange* (see Figures 2 and 3).

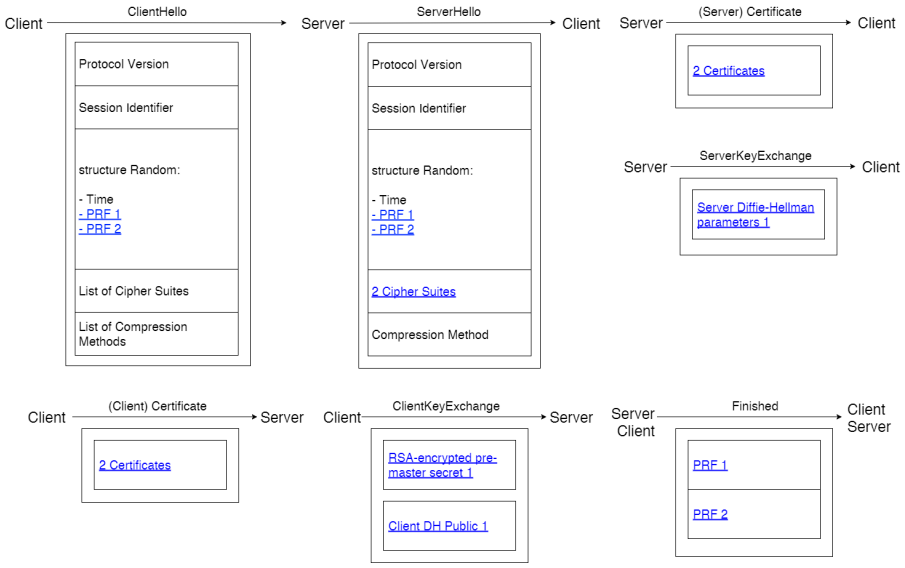

Fig. 2: Example of our proposal's messages using $k = 2$ and choosing two hypothetical cipher suites – TLS_DHE_DSS_WITH_AES_12_CBC_SHA and TLS_RSA_WITH_AES_256_CBC_SHA256. The diversity and redundancy entry points are marked in blue and underlined.

The first message to be sent is called ClientHello. This message's purpose is to inform the server that the client wants to establish a diverse secure channel for communication. The content of this first message consists in the client's protocol version, a structure called Random (analogous to TLS 1.2) containing the current time and a 28-byte pseudo-randomly generated number. TLS uses only one pseudo-random function for the generation of the secure random number. Diversity and redundancy are introduced in this step for the first time. Instead of just one, our solution uses k pseudo-random functions and generates k secure random numbers. It also sends some additional information such as the session identifier, a list of the client's available cipher suites and a list of the client's available compression methods.

The server responds with a message named ServerHello. ServerHello is a very important message as it is where the session's k cipher suites will be agreed-upon. The server sends to the client its protocol version, a structure called Random (identical to the one sent by the client), applying the same diversity, the session identifier, and the k cipher suites chosen by the server from the list the client sent. It is also sent the chosen compression method to use.

The server proceeds to send a Certificate message, containing its certificates, to the client. Each certificate is associated with one KEM. The server must have, and send to the client, i certificates, one certificate for each i different KEM's chosen.

The ServerKeyExchange message is the next message to be sent to the client by the server. It is only sent if one of the k cipher suites includes an ephemeral key mechanism, namely ephemeral Diffie-Hellman (DHE). The content of this message is the server's j Diffie-Hellman parameters, where j is the number of cipher suites using DHE.

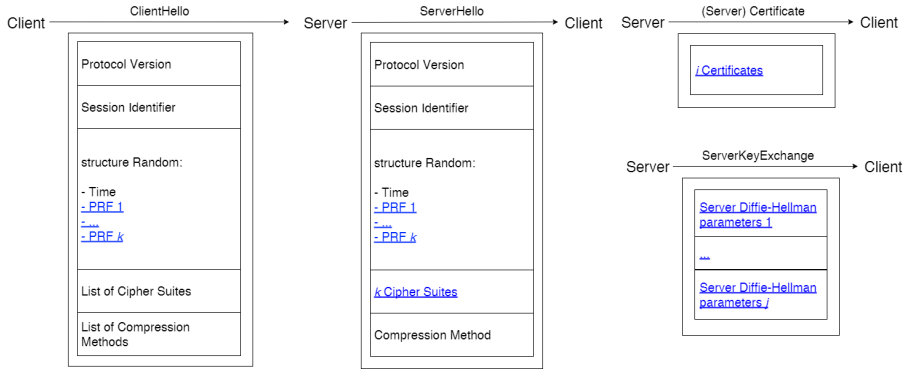

Fig. 3: Our proposal's ClientHello, ServerHello, (Server) Certificate and ServerKeyExchange messages. The diversity and redundancy entry points are marked in blue and underlined.

The remaining messages sent by the server to the client at this point of negotiation, *CertificateRequest* and *ServerHelloDone*, are identical to TLS 1.2 [12].

The client proceeds to send a Certificate message containing its i diverse certificates to the server, analogous to the Certificate message the client received previously from the server. After sending its certificates, the client sends the ClientKeyExchange message. The content of this message is based on the cipher suites chosen. The client sends m RSA-encrypted premaster secrets to the server, with $m \geq 0$. m the subset of the k cipher suites that use RSA for KEM. The client also sends its n Diffie-Hellman public values to the server, where n is the number of cipher suites using Diffie-Hellman. Even if the k cipher suites share the same KEM, this methodology still applies as we can introduce diversity by using different parameters for each KEM (see Fig. 4).

The server may need to verify the client's k certificates. If they have signing capabilities, the client digitally signs all the previous handshake messages and sends them to the server for verification.

Client and server now exchange *ChangeCipherSpec* messages, alike the Cipher Spec Protocol of TLS 1.2, in order to state that they are now using the previously negotiated cipher suites for exchanging messages in a secure fashion.

The client and server, in order to finish the Handshake, send to each other a *Finished* message. This is the first message sent using the k cipher suites negotiated earlier. Its purpose is to each party receive and validate the data received

in this message. If the data is valid, client and server can now exchange messages using the channel. The Finished message uses a pseudo-random function to generate its content. Diversity will also be applied here, using k pseudo-random functions to generate the message's content.

Fig. 4: Our proposal's (Client) Certificate, ClientKeyExchange and Finished messages. The diversity and redundancy entry points are marked in blue and underlined.

5 Evaluation

There will be two main evaluation aspects of our solution – performance and security.

Regarding performance, it will be evaluated in terms of how the increase of complexity in cryptographic mechanisms affects the communication. The time between the moment when a client requests a connection and the secure channel is established will be measured. This is, effectively, the length of the Handshake. After the channel's establishment, the time taken to send a message is also going to be measured. This allows us to assess the overhead generated by using multiple ciphers suites to encrypt a message in opposite to one cipher suite. The overhead, having a k diversity factor, should be, at most, k. Comparisons will be made with TLS 1.2 with different suites.

Having multiple encryption also increases the message size. In order to measure the increase, we will measure the throughput of the channel. This measurement allows us to evaluate the number of messages being sent in a certain period of time. The throughput of our solution is going to, naturally, be inferior to the throughout of a normal TLS 1.2 connection due to the fact that the messages have a bigger size. Additional to measuring the throughput, we will also evaluate how many extra bytes are sent in a message using our solution, comparing to a identical TLS 1.2 message. Using this evaluation, we will be able to state if the additional time taken to send a message is related to the encryption process or related to the message size being significantly bigger.

Overall, the end-user should not experience a slower connection than using TLS 1.2. In the worst case, users should experience a connection k times slower. Even if the communication is proven to be slower, it should be unnoticeable to the end-user.

Additional evaluation may be performed. This additional evaluation regards probabilistic models to evaluate how good and diverse our solution is. Probabilistic models of this kind were proposed by Littlewood *et al.*, in 2004 [27]. These researchers present three models regarding the probability of two systems failing to detect an attack. In our solution, those probabilistic models can be translated to the probability of k systems being vulnerable to an attack. Nevertheless, this evaluation method is based on some speculation:

- The probability of a system being attacked by a certain type of attack;
- The set of all possible attacks;
- The probability of a cipher suite being vulnerable to a certain type of attack;
- The covariance of the k cipher suites, i.e., how similar those two cipher suites are.

This evaluation is not very precise but if the parameters are chosen wisely, we will be able to estimate the probability of our system being vulnerable to a certain type of attacks.

6 Schedule of Future Work

The work schedule is the following:

- January, 9th - March, 15th: Designing and implementing the proposed architecture. Write the corresponding section of the dissertation;
- March, 16th - May, 2nd: Experimental evaluation of the solution and analysis of the results. Write the corresponding section of the dissertation;
- May, 3rd - May, 30th: Write a paper describing the project;
- May, 31st - June, 29th: Write the remaining of the dissertation document;
- June, 30th - Deliver the MSc dissertation.

7 Conclusions

The research was focused in studying several topics concerning the scope of the project. The first one regarded the SSL/TLS protocol and known attacks to its specification, different implementations and supported cryptographic mechanisms. Previous research in diversity and redundancy usage in security and implementations of diversity in various systems were also focused. Finally, vulnerabilities in most used cryptographic mechanisms were also studied and how these cryptographic mechanisms can be combined. We propose to develop a secure communication channel tolerant to vulnerabilities using diversity and redundancy of cryptographic mechanisms.

This work aims to mitigate the problem of secure communication channels relying in one cipher suite and, consequently, incurring the risk of being vulnerable to attacks which makes the channel and the communication insecure. Ultimately, a diverse secure communication channel would be able to resist all attacks, including zero-day vulnerabilities, considering it has k diverse mechanisms which will optimally increase k times the channel's security.

References

1. D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. Vandersloot, E. Wustrow, and S. Paul. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. *22nd ACM Conference on Computer and Communications Security (CCS '15)*, 2015.
2. A. Avižienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the IEEE International Computer Software and Applications Conference*, pages 149–155, 1977.
3. R. Barnes, M. Thomson, A. Pironti, and A. Langley. Deprecating Secure Sockets Layer Version 3.0 (RFC 7568), 2015.
4. B. Beurdouche, K. Bhargavan, A. Delignat-lavaud, C. Fournet, and M. Kohlweiss. A Messy State of the Union: Taming the Composite State Machines of TLS. *IEEE Symposium on Security and Privacy*, pages 1–19, 2015.

5. A. Biryukov, D. Khovratovich, and I. Nikolić. Distinguisher and Related-Key Attack on the Full AES-256. *Advances in Cryptology – CRYPTO 2009*, 5677:231–249, 2009.
6. A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique cryptanalysis of the full AES. *Lecture Notes in Computer Science*, 7073 LNCS:344–371, 2011.
7. J. Callas, L. Donnerhack, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format (RFC 4880), 2007.
8. M. Carvalho, J. DeMott, R. Ford, and D. Wheeler. Heartbleed 101. *IEEE Security & Privacy*, 12(4):63–67, 2014.
9. M. Carvalho and R. Ford. Moving-target defenses for computer networks. *IEEE Security and Privacy*, 12(2):73–76, 2014.
10. S. Choi, J. Herranz, D. Hofheinz, J. Hwang, E. Kiltz, D. Lee, and M. Yung. The Kurosawa–Desmedt key encapsulation is not chosen-ciphertext secure. *Information Processing Letters*, 109(16):897–901, 2009.
11. T. de Raadt. OpenBSD 3.3. http://www.openbsd.org/33.html, 2003. Last time accessed: 2015-10-20.
12. T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol, Version 1.2 (RFC 5246), 2008.
13. W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
14. W. Diffie and M. Hellman. Exhaustive Cryptanalysis of the NBS Data Encryption Standard. *Computer*, 10(6):74–84, June 1977.
15. D. Evans, A. Nguyen-Tuong, and J. Knight. Moving Target Defense. *Moving Target Defense Advances in Information Security*, 54:29–48, 2011.
16. H. Feistel. Data Encryption Standard (DES). *Fips Pub 46-3*, 3, 1999.
17. N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting. Improved cryptanalysis of rijndael. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Bruce Schneier, editors, *Fast Software Encryption*, volume 1978 of *Lecture Notes in Computer Science*, pages 213–230. Springer Berlin Heidelberg, 2001.
18. A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0 (RFC 6101), 2011.
19. E. Fujisaki and T. Okamoto. Secure Integration of Asymmetric and Symmetric Encryption Schemes. *Journal of Cryptology*, 26(1):80–101, 2013.
20. D. Goldschlag, M. Reed, and P. Syverson. Hiding Routing information. *Information Hiding*, 1174:137–150, 1996.
21. 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)*, pages 1–11, 2013.
22. S. Kent and K. Seo. Security Architecture for the Internet Protocol (RFC 4301), 2005.
23. T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. Osvik, H. Te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-Bit RSA modulus. *Lecture Notes in Computer Science*, 6223 LNCS:333–350, 2010.
24. K. Kurosawa and Y. Desmedt. A new paradigm of hybrid encryption scheme. *CRYPTO 2004*, pages 426–442, 2004.
25. A. Langley. The POODLE bites again (08 Dec 2014). https://www.imperialviolet.org/2014/12/08/poodleagain.html, 2014. Last time accessed: 2015-11-13.

26. P. Larsen, S. Brunthaler, and M. Franz. Automatic Software Diversity. *IEEE Security & Privacy*, (April), 2015.
27. B. Littlewood and L. Strigini. Redundancy and Diversity in Security. *Computer Security ESORICS 2004*, pages 227–246, 2004.
28. S. Lucks. Attacking Triple Encryption. *Fast Software Encryption, 5th International Workshop, FSE '98, Paris, France, March 23-25, 1998, Proceedings*, 1372:239–253, 1998.
29. A. Menezes, P. van Oorschot, and S. Vanstone. Hash Functions and Data Integrity. In *Handbook of Applied Cryptography*, chapter 9, pages 321–383. 1996.
30. A. Menezes, P. van Oorschot, and S. Vanstone. Public-Key Encryption. In *Handbook of Applied Cryptography*, chapter 8. CRC Press, Inc., 1996.
31. R. Merkle and M. Hellman. On the security of multiple encryption. *Communications of the ACM*, 24(7):465–467, 1981.
32. R. C. Merkle. *Secrecy, Authentication, and Public Key Systems.* PhD thesis, Stanford, CA, USA, 1979. AAI8001972.
33. B. Moeller and A. Langley. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks (DRAFT). https://tools.ietf.org/html/draft-ietf-tls-downgrade-scsv-00, 2014. Last time accessed: 2015-11-13.
34. B. Möller, T. Duong, and K. Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback. 2014.
35. NIST. Recommendation for Key Management - Part 1: General. *NIST Special Publication 800-57*, Revision 3(July):1–147, 2012.
36. C. Rackoff and D. Simon. Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack. *Advances in Cryptology — CRYPTO '91 SE - 35*, 576:433–444, 1992.
37. V. Rijmen and J. Daemen. Advanced Encryption Standard. *U.S. National Institute of Standards and Technology (NIST)*, 2009:8–12, 2001.
38. R. Rivest. The MD5 Message-Digest Algorithm (RFC 1321), 1992.
39. R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
40. R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.
41. P. Rogaway. Authenticated-Encryption with Associated-Data. (September), 2002.
42. B. Schneier. When Will We See Collisions for SHA-1? https://www.schneier.com/blog/archives/2012/10/when_will_we_se.html, 2012. Last time accessed: 2016-01-03.
43. B. Schneier. SHA-1 Freestart Collision. https://www.schneier.com/blog/archives/2015/10/sha-1_freestart.html, 2015. Last time accessed: 2016-01-03.
44. R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension (RFC 6520), 2012.
45. Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS) (RFC 7457), 2015.
46. P. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Scientific and Statistical Computing*, 26:1484, 1995.
47. M. Stevens. *Attacks on Hash Functions and Applications.* PhD thesis, 2012.
48. M. Stevens, P. Karpman, and T. Peyrin. Freestart collision on full SHA-1. Cryptology ePrint Archive, Report 2015/967, 2015.

49. S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0 (RFC 6176), 2011.
50. F. Valsorda. Logjam: the latest TLS vulnerability explained. https://blog.cloudflare.com/logjam-the-latest-tls-vulnerability-explained/, 2015. Last time accessed: 2015-11-13.
51. X. Wang, Y. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In *Proceedings of the 25th Annual International Conference on Advances in Cryptology*, CRYPTO'05, pages 17–36, Berlin, Heidelberg, 2005. Springer-Verlag.
52. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'05, pages 19–35, Berlin, Heidelberg, 2005. Springer-Verlag.
53. T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture (RFC 4251), 2006.