Vulnerability-Tolerant Transport Layer Security

André Joaquim, Miguel L. Pardal, and Miguel Correia

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
{andre.joaquim, miguel.pardal, miguel.p.correia}@tecnico.ulisboa.pt

Abstract

SSL/TLS communication channels play a very important role in Internet security, including cloud computing and server infrastructures. There are often concerns about the strength of the encryption mechanisms used in TLS channels. Vulnerabilities can lead to some of the cipher suites once thought to be secure to become insecure and no longer recommended for use or in urgent need of a software update. However, the deprecation/update process is very slow and weeks or months can go by before most web servers and clients are protected, and some servers and clients may never be updated. In the meantime, the communications are at risk of being intercepted and tampered by attackers.

In this paper we propose an alternative to TLS to mitigate the problem of secure communication channels being susceptible to attacks due to unexpected vulnerabilities in its mechanisms. Our solution, called Vulnerability-Tolerant Transport Layer Security (vtTLS), is based on diversity and redundancy of cryptographic mechanisms and certificates to ensure a secure communication even when one or more mechanisms are vulnerable. Our solution relies on a combination of k cipher suites which ensure that even if $k-1$ cipher suites are insecure or vulnerable, the remaining cipher suite keeps the communication channel secure. The performance and cost of vtTLS were evaluated and compared with OpenSSL, one of the most widely used implementations of TLS.

1998 ACM Subject Classification C.2.2 Network Protocols; D.4.6 Security and Protection

Keywords and phrases Secure communication channels; Transport layer security; SSL/TLS; Diversity; Redundancy; Vulnerability tolerance

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.23

1 Introduction

Secure communication protocols are extremely important in the Internet. *Transport Layer Security* (TLS) alone is responsible for protecting most economic transactions done using the Internet, with a value too high to estimate. These protocols allow entities to exchange messages or data over a secure channel in the Internet. A secure communication channel has three main properties: *authenticity* – no one can impersonate the sender; *confidentiality* – only the intended receiver of the message is able to read it; and *integrity* – tampered messages can be detected.

Several secure communication channel protocols exist nowadays, with different purposes but with the same goal of securing communication. TLS is a widely used secure channel protocol. Originally called Secure Sockets Layer (SSL), its first 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 in relation to SSL 3.0. TLS 1.1 and TLS 1.2 are upgrades to TLS 1.0 which brought improvements such as mitigation of cipher block chaining (CBC) attacks and supporting more block cipher modes to use with AES.


© James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão;
licensed under Creative Commons License CC-BY

Other widely used secure channel protocols are IPsec and SSH. *Internet Protocol Security* (IPsec) is a network layer protocol that protects the communication at a lower level than SSL/TLS, which operates at the transport layer [18]. IPsec is an extension of the Internet Protocol (IP) that contains two sub-protocols: AH (Authentication Header) that can assure full packet authenticity and ESP (Encapsulating Security Payload) that can protect confidentiality and integrity of the payload of the packets. *Secure Shell* (SSH) is an application-layer protocol used for secure remote login and other secure network services over an insecure network [38].

A secure channel protocol becomes insecure when a vulnerability is discovered. Vulnerabilities may concern the specification of the protocol, the cryptographic mechanisms used, or specific implementations of the protocol. Many vulnerabilities have been discovered in TLS originating new versions of the protocol, deprecating cryptographic mechanisms or enforcing additional security measures. Concrete implementations of TLS have been also found vulnerable due to implementation bugs. The processes of deprecation or software update are very slow and can take a long time to become effective [2]. Also, they may not even reach all the affected servers and clients. This means that the communications between devices are at risk of interception or tampering by attackers for a long period.

This paper proposes a *vulnerability-tolerant* communication channel protocol, based on TLS, named Vulnerability-Tolerant Transport Layer Security (vtTLS). A vulnerability-tolerant channel is characterized by not relying on individual cryptographic mechanisms, so that if any one of them is found vulnerable, the channel still remains secure. The idea is to leverage *diversity* and *redundancy* of cryptographic mechanisms and keys by using more than one set of mechanisms/keys. This use of diversity and redundancy is inspired in previous works on intrusion tolerance [34], diversity in security [21, 13] and moving-target defenses [7].

Consider for example SHA-1 and SHA-3, two hash functions that may be used to generate message digests. If used in combination and SHA-1 eventually becomes insecure, vtTLS would rely upon SHA-3 to keep the communication secure.

vtTLS is configured with a parameter k ($k > 1$), the *diversity factor*, that indicates the number of different cipher suites and different mechanisms for key exchange, authentication, encryption, and signing. This parameter means also that vtTLS remains secure as long as less than k vulnerabilities exist. As vulnerabilities and, more importantly, zero-day vulnerabilities cannot be removed as they are unknown [4], do not appear in large numbers in the same components, we expect k to be usually small, e.g., $k = 2$ or $k = 3$.

Although TLS supports strong encryption mechanisms such as AES and RSA, there are factors beyond mathematical complexity that can contribute to vulnerabilities. Diversifying encryption mechanisms includes diversifying certificates and consequently keys (public, private, shared). Diversity of certificates is a direct consequence of diversifying encryption mechanisms due to the fact that each certificate is related to an authentication and key exchange mechanism.

The main contribution of this paper is vtTLS, a new protocol for secure communication channels that uses diversity and redundancy to tolerate vulnerabilities in cryptographic mechanisms. The experimental evaluation shows that vtTLS has an acceptable overhead in relation to the TLS implementation in OpenSSL v1.0.2g [35].

The rest of the document is organized as follows. Section 2 presents background and related work. Section 3 presents the protocol and Section 4 its implementation. Section 5 presents the experimental evaluation and Section 6 concludes the paper.

2 Background and Related Work

This section presents related work on diversity (and redundancy) in security, provides background information on TLS, and discusses vulnerabilities in cryptographic mechanisms and protocols.

2.1 Diversity

The term *diversity* is used to describe multiple version software in which redundant versions are deliberately created and made different between themselves [21]. Without diversity, all instances are the same, with the same implementation vulnerabilities. Using diversity it is possible to present the attacker with different versions, hopefully with different vulnerabilities. Software diversity targets mostly software implementation and the ability of the attacker to replicate the user's environment. Diversity does not change the program's logic, so it is not helpful if a program is badly designed. According to Littlewood and Strigini, multi-version systems on average are more reliable than those with a single version [21]. They also state that the key to achieve effective diversity is to make the dependence between the different programs as low as possible. Therefore, attention is needed when choosing the diverse versions. The trade-off between individual quality and dependence needs to be assessed and evaluated, as it impacts the correlation between version failures.

Recently there has been some discussion on the need for moving-target defenses. Such defenses dynamically alter properties of programs and systems in order to overcome vulnerabilities that eventually appear in static defense mechanisms [11]. There are two types of moving-target defenses: proactive and reactive [7]. Proactive defenses are generally slower than reactive defenses as they prevent attacks by increasing the system complexity periodically. Reactive defenses are faster 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 not detected. In this case, reactive defenses are worthless, but proactive defenses may prevent that attack from being successful. The best approach would be to implement both [30]. Nevertheless, these defences are as good as their ability to make an unpredictable change to the system.

Earlier, Avižienis and Chen introduced N-version programming (NVP) [3]. NVP is defined as the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification. The authors state that in order to use redundant programs to achieve fault tolerance the redundant program must contain independently developed alternatives routines for the same functions. The N in NVP comes from N diverse versions of a program developed by N different programmers, that do not interact with each other regarding the programming process. One of the limitations of NVP is that every version is originated from the same initial specification. There is the need to assure the initial specification's correctness, completeness and unambiguity prior to the versions development.

There is some work on obtaining diversity without explicitly developing N versions [20]. Garcia *et al.* show that there is enough diversity among operating systems for several practical purposes [13]. Homescu *et al.* use profile-guided optimization for automated software diversity generation [15]. Transparent Runtime Randomization (TRR) dynamically and randomly relocates parts of the program to provide different versions [37]. Proactive obfuscation aims to generate replicas with different vulnerabilities [26]. We introduced the idea of using diversity for vulnerability-tolerant channels in a short (4-page) paper [16]. The present paper greatly expands that previous work.

2.2 TLS Protocol

TLS has two main sub-protocols: the Handshake Protocol and the Record Protocol.

The Handshake Protocol is 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.

The Record protocol is the sub-protocol which processes the messages to send and receive after the handshake, i.e., in normal operation. Regarding an outgoing message, the first operation performed by the Record protocol is fragmentation. After fragmenting the message, each block may be optionally compressed, using the compression method defined in the Handshake. Each potentially compressed block is now transformed into a ciphertext block by encryption and message authentication code (MAC) functions. Each ciphertext 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. When using block ciphers, padding and its length is added 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 (MAC-then-Encrypt mode using a SHA-2 variant), no MAC key is used. The message is then sent to its destination. Regarding an incoming message, the process is the inverse. The message is decrypted, verified, optionally decompressed, reassembled and delivered to the application.

2.3 TLS Vulnerabilities

We now discuss some of the vulnerabilities discovered in TLS in the past to show the relevance of our work to bring added security to communications. TLS vulnerabilities can be classified in two types: specification and implementation. *Specification vulnerabilities* concern the protocol itself. A specification vulnerability can only be fixed by a new protocol version or an extension. *Implementation vulnerabilities* exist in the code of an implementation of TLS, such as OpenSSL. This section presents some of the most recent [28].

An example attack that exploits a specification vulnerability is *Logjam* [1]. The attack consists in exploiting several weak parameters in the Diffie-Hellman key exchange. Logjam is a man-in-the-middle attack that downgrades the connection to a weakened Diffie-Hellman mode. This man-in-the-middle attack changes the cipher suites used in the `DHE_EXPORT` cipher suite, forcing the use of weaker Diffie-Hellman key exchange parameters. As the server supports this 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. The client sees that the server has chosen a seemingly normal DHE option and proceeds to compute its secret also with weak parameters. At 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.

Heartbleed is an example of an *implementation vulnerability*. It was a bug in C code that existed in OpenSSL 1.0.1 through 1.0.1f, when the heartbeat extension was introduced and enabled by default [27]. The Heartbleed vulnerability allowed an attacker to perform a buffer over-read, reading up to 64 KB from the memory of the victim [6].

2.4 Vulnerabilities in Cryptographic Schemes

This section presents example vulnerabilities in cryptographic mechanisms used by TLS.

2.4.1 Public-key Cryptography

RSA is a widely-used public-key cryptography scheme. Its security is based on the difficulty of factorization of large integers and the RSA problem [23]. RSA can be considered to be broken if these problems can be solved in a practical amount of time.

Kleinjung *et al.* performed the factorization of RSA-768, a RSA number with 232 digits [19]. The researchers state they spent almost two years in the whole process, which is clearly a non-practical time. Factorizing a large integer is different from breaking RSA, which is still secure. As of 2010, these researchers concluded that RSA-1024 would be factored within five years. As for now, no factorization of RSA-1024 has been publicly announced, but key sizes of 2048 and 3072 bits are now recommended [10].

Shor designed a quantum computing algorithm to factorize integers in polynomial time [29]. However, it requires a quantum computer able to run it, which is still not publicly available.

2.4.2 Symmetric Encryption

The Advanced Encryption Standard (AES), originally called Rijndael, is the current American standard for symmetric encryption [25]. 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 is used by many protocols, including TLS.

The most successful cryptanalysis of AES was published by Bogdanov *et al.* in 2011, using a biclique attack, a variant of the MITM attack [5]. 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. Ferguson *et al.* presented the first known attacks on the first seven and eight rounds of Rijndael [12]. Although it shows some advance in breaking AES, AES with a key of 128 bits has 10 rounds.

2.4.3 Hash Functions

The main uses for hash functions are data integrity and message authentication. A hash, also called message digest or digital fingerprint, is a compact representation of the input and can be used to uniquely identify that same input [22]. 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.

The Secure Hash Algorithm 1 (SHA-1) is a cryptographic hash function that produces a 160-bit message digest. Its use is not recommended for some years [10], although the first collision was discovered only recently [32]. There have been some previous attacks against SHA-1. Stevens *et al.* presented a freestart collision attack for SHA-1's internal compression function [33]. Taking into consideration the Damgard-Merkle [24] 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). 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} [36]. 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 2011, Stevens improved the number of calculations needed to produce a collision from 2^{69} to a number between $2^{60.3}$ and $2^{65.3}$ [31].

3 Vulnerability-Tolerant TLS

vtTLS is a new protocol that provides vulnerability-tolerant secure communication channels. It aims at increasing security using diverse and redundant cryptographic mechanisms and certificates. It is based on the TLS protocol. The protocol aims to solve the main problem originated by having only one cipher suite negotiated between client and server: when one of the cipher suite's mechanisms becomes insecure, the communication channels using that cipher suite may become vulnerable. Although most cipher suites' cryptographic mechanisms supported by TLS 1.2 are believed to be secure, Section 2 shows clearly that new vulnerabilities may be discovered.

Unlike TLS, a vtTLS communication channel does not rely on only one cipher suite. vtTLS negotiates more than one cipher suite between client and server and, consequently, more than one cryptographic mechanism will be used for each *phase*: key exchange, authentication, encryption and MAC. Diversity and redundancy appear firstly in vtTLS in the Handshake protocol, in which client and server negotiate k cipher suites to secure the communication, with $k > 1$.

The strength of vtTLS resides in the fact that even when $(k-1)$ cipher suites become insecure, e.g., because $(k-1)$ of the cryptographic mechanisms are vulnerable, the protocol remains secure. The server chooses the best combination of k cipher suites according to the cipher suites server and client have available. However, the choice of the cipher suites might be conditioned by the certificates of both server and client. vtTLS uses a subset of the k cipher suites agreed-upon in the Handshake Protocol to encrypt the messages.

3.1 Protocol Specification

The vtTLS Handshake Protocol is similar to the TLS Handshake Protocol. We use the same names for the messages in order to help the reader familiarized with TLS.

The messages that require diversity are ClientHello, ServerHello, k-ServerKeyExchange, Server and Client Certificate, and k-ClientKeyExchange.

The first message to be sent is ClientHello to inform the server that the client wants to establish a secure channel for communication.

The server responds with a ServerHello message. This is where the server sends to the client the k cipher suites to be used in the communication. The server also sends its protocol version, a Random structure identical to the one received from the client, the session identifier, and the k cipher suites chosen by the server from the list the client sent.

The server proceeds to send a Server Certificate message containing its k certificates to the client. The k chosen cipher suites are dependent from the server's certificates. Each certificate is associated with one key exchange mechanism (KEM). Therefore, the k cipher suites must use the key exchange mechanisms supported by the server's certificates.

vtTLS behaves correctly if the server has c certificates, with $0 < c \leq k$. The cipher suites to be used are chosen considering the available certificates. If $c < k$, the diversity is not fully achieved due to the fact that a number of cipher suites will share the same key exchange and authentication mechanisms.

The ServerKeyExchange message is the next message to be sent to the client by the server. This message is only sent if one of the k cipher suites includes a key exchange mechanism like ECDHE or DHE that uses ephemeral keys, i.e., that generate new keys for every key exchange. The contents of this message are the server's DH ephemeral parameters. For every other $k-1$ cipher suites using ECDHE or DHE, the server sends additional ServerKeyExchange messages with additional diverse DH ephemeral parameters. Instead

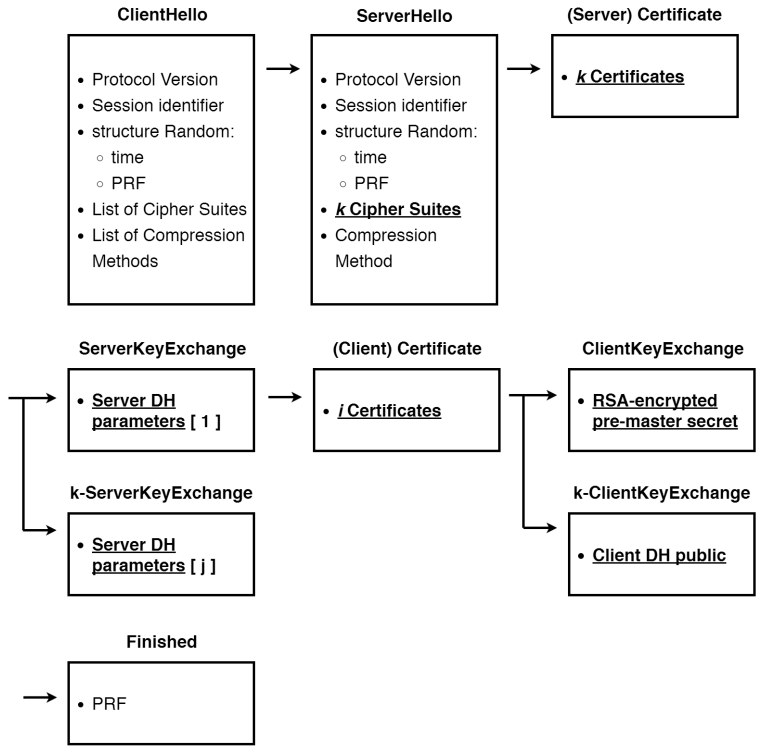

Figure 1 vtTLS Handshake messages with diversity factor k. The places where diversity and redundancy are introduced are marked in bold and underlined.

of computing all the ephemeral parameters and sending them all on a single larger message, the server, after computing one parameter, sends it immediately, sending each parameter in a separate message.

The remaining messages sent by the server to the client at this point of the negotiation, CertificateRequest and ServerHelloDone, are identical to those in TLS 1.2 [9].

The client proceeds to send a (Client) Certificate message containing its i certificates to the server, analogous to the (Server) Certificate message the client received previously from the server.

After sending its certificates, the client sends k ClientKeyExchange messages to the server. The content of these messages is based on the k cipher suites chosen. If m of the cipher suites use RSA as KEM, the client sends m messages, each one with a RSA-encrypted pre-master secret to the server ($0 \leq m \leq k$). If j of the cipher suites use ECDHE or DHE, the client sends j messages to the server containing its j Diffie-Hellman public values ($0 \leq j \leq k$). Even if a subset of the k cipher suites share the same KEM, this methodology still applies as we introduce diversity by using different parameters for each cipher suite being used.

The server needs to verify the client's i certificates. The client digitally signs all the previous handshake messages and sends them to the server for verification.

Client and server now exchange ChangeCipherSpec messages, like in 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.

In order to finish the Handshake, the client and server send each other a Finished message. This is the first message sent encrypted using the k cipher suites negotiated earlier.

Its purpose is for each party to receive and validate the data received in this message. If the data is valid, client and server can now exchange messages over the communication channel.

3.2 Combining Diverse Cipher Suites

Diversity between cryptographic mechanisms can be taken in a soft sense as the use of different mechanisms, or in a hard sense as the use of mechanisms that do not share common vulnerabilities (e.g., because they are based on different mathematical problems). In vtTLS we are interested in using strong diversity in order to claim that no common vulnerabilities will appear in different mechanisms. Measuring the level of diversity is not simple, so we leverage previous research by Carvalho on heuristics for comparing diversity among cryptographic mechanisms [8]. Moreover, not all cryptographic mechanisms can be used together in the context of TLS 1.2 and other security protocols. Here we consider only the combinations of two algorithms, i.e., $k = 2$, for simplicity.

After comparing several hash functions, Carvalho concluded that the best three combinations are the following:

- SHA-1 + SHA-3: not possible in vtTLS as SHA-1 is not recommended and TLS 1.2 does not support SHA-3;
- SHA-1 + Whirlpool: not possible in vtTLS as SHA-1 is not recommended and TLS 1.2 does not support Whirlpool;
- SHA-2 + SHA-3: also not possible in vtTLS as TLS 1.2 does not support SHA-3.

All the remaining combinations suggested in that work cannot also be used because TLS 1.2 does not support SHA-3. All vtTLS cipher suites use either AEAD or SHA-2 (SHA-256 or SHA-384). Having a small range of available hash functions limits the maximum diversity factor achievable concerning hash functions. In a near future, it is expected that a new TLS protocol version supports SHA-3 and makes possible the use of diverse hash functions. Nevertheless, it still possible to achieve diversity by using different variants of SHA-2: SHA-256 and SHA-384.

After comparing several public-key encryption mechanisms, Carvalho concluded that the best four combinations are:

- DSA + RSA: possible as TLS 1.2 supports both functions for *authentication*. However, TLS 1.2 specific cipher suites only support DSA with elliptic curves (ECDSA);
- DSA + Rabin-Williams: not possible as TLS 1.2 does not support Rabin-Williams;
- RSA + ECDH: possible as TLS 1.2 supports both functions for *key exchange*;
- RSA + ECDSA: possible as TLS 1.2 supports both functions for *authentication*.

Regarding authentication, although DSA + RSA is stated as the most diverse combination, TLS 1.2 preferred cipher suites use ECDSA instead of DSA. Using elliptic curves results in a faster computation and lower power consumption [14]. With that being said, the preferred combination for authentication is RSA + ECDSA.

Regarding key exchange, the most diverse combination is RSA + ECDH. However, in order to grant perfect forward secrecy, the ECDH with ephemeral keys (ECDHE) has to be employed. Concluding, the preferred combination for key exchange is RSA + ECDHE.

The study in [8] did not present any conclusions regarding symmetric-key encryption. However, both AES and Camellia are supported by TLS 1.2 and are considered secure. The most diverse combination is AES256-GCM + CAMELLIA128-CBC: the origin of the two algorithms is different, they were first published in different years, they both have semantic security (as they both use initialization vectors) and the mode of operation is also different. One constraint of using this combination is that there is no cipher suite that uses RSA for

key exchange, Camellia for encryption and a SHA-2 variant for MAC. Although RFC 6367 [17] describes the support for Camellia HMAC-based cipher suites, extending TLS 1.2, these cipher suites are not supported by OpenSSL 1.0.2g. Using a cipher suite that uses Camellia, in order to maximize diversity, implies using also SHA-1 for MAC and not using ECDHE for key exchange nor ECDSA for authentication in that cipher suite. Concluding, using Camellia increases diversity in encryption but reduces security in MAC, forcing the use of an insecure algorithm. Nevertheless, diversity in encryption is still an objective to accomplish. We decided that the best option is:

- AES256-GCM + AES128: possible as TLS 1.2 supports both functions.

These functions are, in theory, the same, but employed with a different strength size and mode of operation, they can be considered diverse, although they have an inferior degree of diversity comparing to any of the combinations above.

Concluding, the best combination of cipher suites is arguably:

- `TLS_ECDHE_ECDSA_WITH_AES_ 256_GCM_SHA384` and
- `TLS_RSA_WITH_AES_128_CBC_SHA256`

For key exchange, vtTLS will use Ephemeral ECDH (ECDHE) and RSA; for authentication, it will use Elliptic Curve DSA (ECDSA) and RSA; for encryption, it will use AES-256 with Galois/Counter mode (GCM) and AES-128 with cipher block chaining (CBC) mode; finally, for MAC, it will use SHA-2 variants (SHA-384 and SHA-256). Using this combination of cipher suites, the lowest diversity is with the MAC, due to the fact that TLS 1.2 does not support SHA-3 for now.

4 Implementation

Our implementation of vtTLS was obtained by modifying OpenSSL version 1.0.2g.[1]. Implementing a vtTLS from scratch would be a bad option as it might lead to the creation of vulnerabilities; existing software such as OpenSSL has the advantage of being extensively debugged, although serious vulnerabilities like Heartbleed still appear from time to time. Furthermore, creating a new secure communication protocol, and consequently a new API, would create adoption barriers to programmers otherwise willing to use our protocol. Therefore, we chose to implement vtTLS based on OpenSSL, keeping the same API as far as possible. Although being based on OpenSSL, vtTLS is not fully compatible with it due to its diversity and redundancy features. It is noteworthy that OpenSSL is a huge code base (438,841 lines of code in version 1.0.2g) so modifying it to support diversity has been a considerable engineering challenge.

In order to establish a vtTLS communication channel, additional functions are required to fulfill the requirements of vtTLS, such as loading two certificates and corresponding private keys. These functions have a similar name of the ones belonging to the OpenSSL API, to reduce the learning curve. The most relevant functions regarding the setup of the channel are the functions that allow to load the second certificate and private key and allow to check if the second private key corresponds to the second certificate.

Regarding the *Handshake Protocol*, we opted for sending k ServerKeyExchange and ClientKeyExchange messages instead of sending one single ServerKeyExchange and

[1] `https://www.openssl.org`

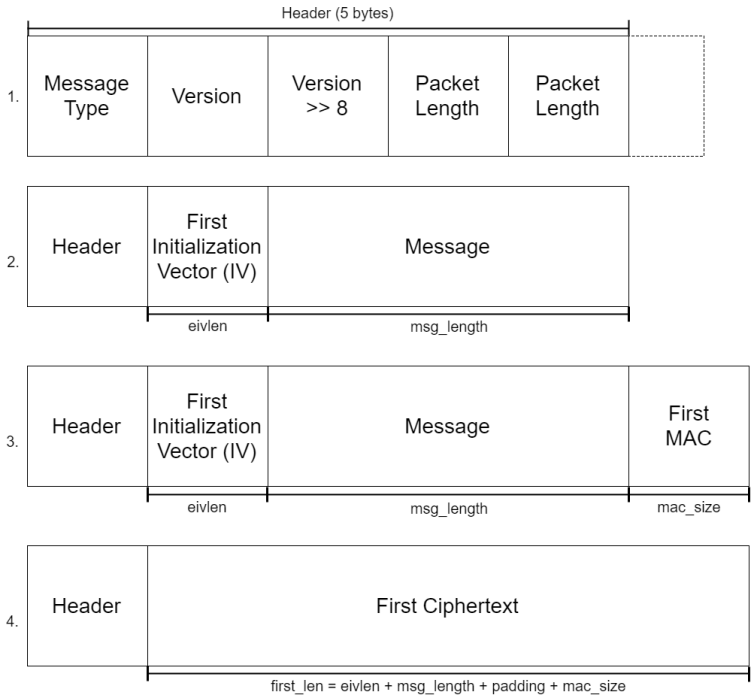

Figure 2 First four steps in the creation of a data message in vtTLS with $k = 2$: first encryption and MAC.

one single ClientKeyExchange, each one with several parameters. This is due to the fact that it makes the code easier to understand and to maintain. If k needs to be increased, it is just needed to send an additional message instead of changing the code related to sending and retrieving ServerKeyExchange and ClientKeyExchange messages.

The encryption and signing ordering is also important in vtTLS. Figure 2 shows the initial steps of the creation of a data message. The figure considers $k = 2$, but shows only the steps regarding the first encryption and the first signature.

Figure 3 shows the final steps of the preparation of a vtTLS message. We opted for maintaining the creation of the MAC prior to the encryption. Using this approach, both message and MACs are encrypted with both ciphers. In this case there is no chance that both MACs are identical, if the hash function used is secure (SHA-2 is considered secure).

The whole sequence is the following:

- Apply the first MAC to the plaintext message;
- Encrypt the original message and its MAC with the first encryption function;
- Apply the second MAC to the first ciphertext;
- Encrypt the first ciphertext and its MAC with the second encryption function.

In relation to the *Record Protocol*, signing and encrypting k times has a cost in terms of message size. Figures 2 and 3 show also the expected increase of the message size due to the use of a second MAC and a second encryption function (for $k = 2$). For TLS 1.2 (OpenSSL), the expected size of a message is $first\_len = eivlen + msg\_length + padding + mac\_size$, where *eivlen* is the size of the initialization vector (IV), *msg_length* the original message size, *padding* the size of the padding in case a block cipher is used, and *mac_size* the size of the MAC (Figure 2). For vtTLS, the additional size of the message is $eivlen\_sec + first\_len +$

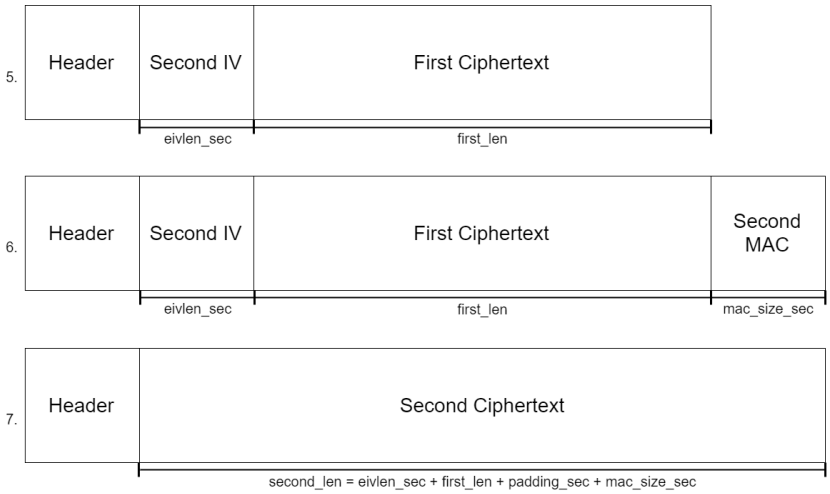

Figure 3 Remaining three steps in the creation of a data message in vTTLS with $k = 2$: second encryption and MAC.

padding_sec + *mac_size_sec*, where *eivlen_sec* is the size of the IV associated with the second cipher and *mac_size_sec* the size of the second MAC.

In the best case, the number of packets is the same for OpenSSL and vTTLS. In the worst case, one additional packet may be sent if the encryption function requires fixed block size and the maximum size of the packet, after the second MAC and the second encryption, is exceeded by, at least, one byte. In this case, an additional full packet is needed due to the constraint of having fixed block size.

5 Evaluation

We evaluated vTTLS in terms of two aspects: *performance* and *cost*. We considered OpenSSL 1.0.2g as the baseline, due to the fact that vTTLS is based on that software and version.

Diversity has performance costs and creates overhead in the communication. Every message sent needs to be ciphered and signed $k-1$ times more than using a TLS implementation and every message received needs to be deciphered and verified also $k-1$ times more. In the worst case, users should experience a connection k times slower than using OpenSSL. We considered $k = 2$ in all experiments, as this is the value we expect to be used in practice (we expect vulnerabilities to appear rarely, so the ability to tolerate one vulnerability per mechanism sufficient). With this experimental evaluation, we want to be able to state if vTTLS is a viable mechanism for daily usage, i.e., if the penalty for replacing TLS channels by vTTLS channels is not prohibitive.

In order to perform these tests, we used two virtual machines in the same Intel Core i7 computer with 8 GB RAM. The virtual machines run Debian 8 and openSUSE 12 playing the roles of server and client, respectively. All the tests were done in the same controlled environment and same geographic location.

5.1 Performance

In order to evaluate the performance of vTTLS, we executed several tests. The main goal was to understand if the overhead of vTTLS is lower, equal, or bigger than k times in relation to OpenSSL. We configured vTTLS to use the following cipher suites:

	Average (ms)	Standard deviation	Confidence interval (95%)
vtTLS	3.909	0.963	±0.180
OpenSSL	2.345	0.933	±0.174

Table 1 Handshake time comparison

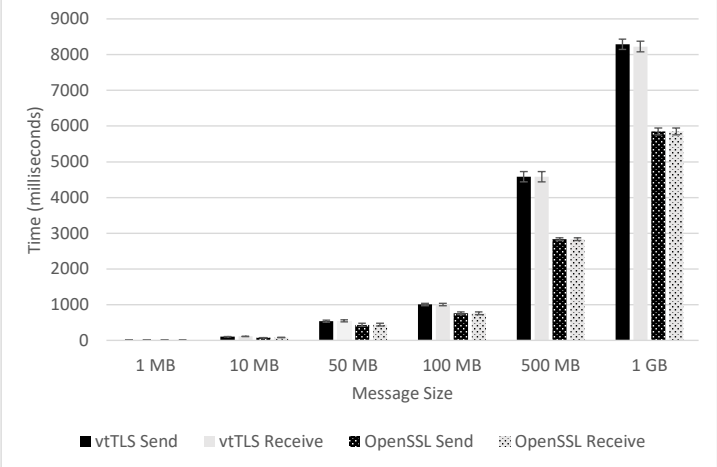

Figure 4 Time to send and receive a message with vtTLS and OpenSSL.

`TLS_RSA_WITH_AES_256_GCM_SHA384` and `TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384`. The suite used with OpenSSL was the latter.

To evaluate the performance of the handshake, we executed 100 times the Handshake Protocol of both vtTLS and OpenSSL. In average, the vtTLS handshake took 3.909 milliseconds to conclude and the OpenSSL handshake 2.345 milliseconds. Therefore, the vtTLS handshake is only 1.67 slower than the OpenSSL handshake, which is better than the worst case. Table 1 provides more details.

After evaluating the Handshake, we performed data communication tests to assess the overhead generated by the diversity and redundancy of mechanisms. As the Handshake, the communication is expected to be at most $k = 2$ times slower than a TLS communication. For this test, we considered a sample of 100 messages sent and received with vtTLS and 100 messages sent and received with OpenSSL.

Figure 4 shows the comparison between the time it takes to send and receive a message with vtTLS and OpenSSL/TLS. Tables 2 and 3 show more details.

The measurements concern the time each channel needs to perform the local operations

	vtTLS Send			OpenSSL Send		
	Average	St. dev.	Conf. I. (95%)	Average	St. dev.	Conf. I. (95%)
1 MB	12.80	3.324	±0.652	11.28	3.985	±0.781
10 MB	105.89	17.573	±3.444	76.61	10.413	±2.041
50 MB	534.55	149.697	±29.340	435.01	212.065	±41.564
100 MB	1004.30	194.701	±38.161	757.02	206.709	±40.514
500 MB	4579.40	727.519	±142.591	2834.18	217.378	±42.605
1 GB	8289.78	757.167	±148.402	5851.08	480.423	±94.161

Table 2 Time to send a message with vtTLS and OpenSSL.

	vtTLS Receive			OpenSSL Receive		
	Average	**St. dev.**	**Conf. I. (95%)**	**Average**	**St. dev.**	**Conf. I.(95%)**
1 MB	14.70	3.324	±0.652	13.48	3.985	±0.781
10 MB	113.41	17.573	±3.444	80.42	10.413	±2.041
50 MB	549.61	149.697	±29.340	443.10	212.065	±41.564
100 MB	1004.54	194.701	±38.161	757.13	206.709	±40.514
500 MB	4580.13	727.519	±142.591	2834.37	217.378	±42.605
1 GB	8227	757.167	±148.402	5850.96	480.423	±94.161

Table 3 Time to receive a message with vtTLS and OpenSSL.

Message size	**vtTLS**		**OpenSSL**		**Overhead (diff.)**	
	Encrypted message size	**Average #packs**	**Encrypted message size**	**Average #packs**	**Packets**	**Message size**
100,000	102,771	6.3	102,603	5.3	1	168
1,000,000	1,029,054	38.3	1,025,856	37.6	0.7	3,198
100,000,000	105,362,077.10	2,830.2	104,956,194.50	2,553.5	276.7	405,883

Table 4 Message sizes in vtTLS and OpenSSL.

in order to send the message (including encryption, signing, second encryption and second signing). These values do not include the time taken by the message to reach its destination through the network. The timer is started before the call to `SSL_write` and stopped after the function returns. As for the results regarding the reception of messages, the measured time is the time taken to perform the operations necessary to retrieve the message (including second decrypting and second verifying), i.e. the time of execution of `SSL_read`. This methodology is only possible due to the fact that `SSL_write` is a synchronous call. It only returns after writing the message to the buffer. And also due to the fact that `SSL_read` is also synchronous as it only returns when the message is read.

In average, a message sent through a vtTLS channel takes 22.88% longer than a message sent with OpenSSL. For example, a 50 MB message takes an average of 534.55 ms to be sent with vtTLS. With OpenSSL, the same message takes 435.01 ms to be sent. The overhead generated by using diverse encryption and MAC mechanisms exists, as expected, but it is much smaller than the expected worst case.

In order to validate the premise that the message increase is the same considering the same message size, we measured the increase in the message size comparing once again vtTLS and OpenSSL channels. A 100 KB plaintext message converts into a ciphertext of 102,771 bytes with vtTLS. With OpenSSL, the same message corresponds to a ciphertext of 102,603 bytes. Concluding, sending a 100 KB message through vtTLS costs an additional 168 bytes. Therefore, as stated before, the number of extra bytes sent is not directly proportional to the message size.

We also evaluated the message size of the ciphertext of a 1 MB plaintext message. A 1 MB plaintext message corresponds to a ciphertext of 1,029,054 bytes using vtTLS, while using OpenSSL the same message has 1,025,856 bytes. Concluding, sending 1 MB through a vtTLS channel costs an additional 3,198 bytes than using a OpenSSL channel. Table 4 shows all the results obtained and the comparison between the message sizes.

5.2 Cost

Similarly to TLS, vtTLS uses certificates that require some management effort and costs. A server using OpenSSL/ TLS to protect the communication with clients needs only one certificate. If the administrator decides to use vtTLS instead of TLS, at least 2 certificates are needed for maximum diversity, and at most k certificates. Although certificates are not expensive, they represent a cost. vtTLS can be used with just one certificate, but this reduces the diversity and, consequently, the potential security benefit.

Regarding management, there is the need to manage two certificates instead of one. We believe this does not represent a substantial increase in management effort. If it is decided to use vtTLS with a diversity factor $k > 2$, the management costs of maintaining k certificates might represent an significant increase of management costs.

6 Conclusions

vtTLS is a diverse and redundant vulnerability-tolerant secure communication protocol designed for communication on the Internet. It aims at increasing security using diverse cipher suites to tolerate vulnerabilities in the encryption mechanisms used in the communication channel. In order to evaluate our solution, we compared it to an OpenSSL 1.0.2g communication channel. While expected to be $k = 2$ times slower than an OpenSSL channel, the evaluation showed that using diversity and redundancy of cryptographic mechanisms in vtTLS does not generate such a high overhead. vtTLS takes, in average, 22.88% longer to send a message than TLS/OpenSSL, but considering the increase in security, this overhead is acceptable. Overall, considering the additional costs of having an extra certificate, the time increase, and potential management costs, vtTLS provides an interesting trade-off for a set of critical applications.

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

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. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 5–1, October 2015.

2 M. R. Albrecht, J. P. Degabriele, T. B. Hansen, and K. G. Paterson. A surfeit of SSH cipher suites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1480–1491, 2016.

3 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.

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

5 A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique cryptanalysis of the full AES. In *Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security*, volume LNCS 7073, pages 344–371, 2011.

6 M. Carvalho, J. DeMott, R. Ford, and D. Wheeler. Heartbleed 101. *IEEE Security & Privacy*, 12(4):63–67, 2014.

7 M. Carvalho and R. Ford. Moving-target defenses for computer networks. *IEEE Security and Privacy*, 12(2):73–76, 2014.

8 R. Carvalho. Authentication security through diversity and redundancy for cloud computing. Master's thesis, Instituto Superior Técnico, Lisbon, Portugal, 2014.

9 T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol, Version 1.2 (RFC 5246), 2008.

10 ENISA. Algorithms, key size and parameters report – 2014. nov 2014.

11 D. Evans, A. Nguyen-Tuong, and J. Knight. Effectiveness of moving target defenses. In *Moving Target Defense*, volume 54, pages 29–48. Springer, 2011.

12 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, *Proceedings of Fast Software Encryption*, volume LNCS 1978, pages 213–230. Springer, 2001.

13 M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, pages 383 –394, 27–30 June 2011.

14 V. Gupta, S. Gupta, S. Chang, and D. Stebila. Performance analysis of elliptic curve cryptography for SSL. In *Proceedings of the 1st ACM Workshop on Wireless Security*, pages 87–94, 2002.

15 A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 1–11, 2013.

16 A. Joaquim, M. L. Pardal, and M. Correia. vtTLS: A vulnerability-tolerant communication protocol. In *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications*, pages 212–215, 2016.

17 S. Kanno and M. Kanda. Addition of the Camellia cipher suites to transport layer security (TLS) (RFC 6367), 2011.

18 S. Kent and K. Seo. Security architecture for the internet protocol (RFC 4301), 2005.

19 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. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, volume LNCS 6223, pages 333–350, 2010.

20 P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 276–291, 2014.

21 B. Littlewood and L. Strigini. Redundancy and diversity in security. In *Computer Security – ESORICS 2004, 9th European Symposium on Research Computer Security*, pages 227–246, 2004.

22 A. Menezes, P. van Oorschot, and S. Vanstone. Hash functions and data integrity. In *Handbook of Applied Cryptography*, chapter 9. CRC Press, 1996.

23 A. Menezes, P. van Oorschot, and S. Vanstone. Public-key encryption. In *Handbook of Applied Cryptography*, chapter 8. CRC Press, 1996.

24 R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford, CA, USA, 1979.

25 V. Rijmen and J. Daemen. Advanced encryption standard. *U.S. National Institute of Standards and Technology (NIST)*, 2009:8–12, 2001.

26 T. Roeder and F. B. Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems*, 28:4:1–4:54, July 2010.

27 R. Seggelmann, M. Tuexen, and M. Williams. Transport layer security (TLS) and datagram transport layer security (DTLS) heartbeat extension (RFC 6520), 2012.

28 Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS) (RFC 7457), 2015.

29 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.

30 P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, April 2010.

31 M. Stevens. *Attacks on Hash Functions and Applications*. PhD thesis, Mathematical Institute, Leiden University, 2012.

32 M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full SHA-1. *IACR Cryptology ePrint Archive*, 2017:190, 2017.

33 M. Stevens, P. Karpman, and T. Peyrin. Freestart collision on full SHA-1. Cryptology ePrint Archive, Report 2015/967, 2015.

34 P. Verissimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677, pages 3–36. 2003.

35 J. Viega, M. Messier, and P. Chandra. *Network Security with OpenSSL: Cryptography for Secure Communications*. O'Reilly, 2002.

36 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*, pages 17–36. Springer-Verlag, 2005.

37 J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 260–269, 2003.

38 T. Ylonen and C. Lonvick. The secure shell (SSH) protocol architecture (RFC 4251), 2006.