vtTLS: A Vulnerability-Tolerant Communication Protocol

ABSTRACT

We present vtTLS, a vulnerability-tolerant communication protocol based on diversity and redundancy. There are often concerns about the strength of some of the encryption mechanisms used in SSL/TLS channels, with some regarded as insecure at some point in time. vtTLS is our solution to mitigate the problem of secure communication channels being vulnerable to attacks due to unexpected vulnerabilities in encryption mechanisms. It is based on diversity and redundancy of cryptographic mechanisms and certificates to provide a secure communication channel even when one or more mechanisms are vulnerable. vtTLS relies on a combination of k cipher suites. Even if $k-1$ cipher suites are insecure or vulnerable, vtTLS relies on the remaining cipher suite to maintain the channel secure. We evaluated the performance of vtTLS by comparing it to an OpenSSL channel.

CCS Concepts

•**Networks** → **Application layer protocols;** *Network security;* •**Computer systems organization** → **Redundancy;** •**Security and privacy** → *Security protocols;* Symmetric cryptography and hash functions;

Keywords

Protocol; Secure communication channels; Diversity; Redundancy; Vulnerability-Tolerance

1. INTRODUCTION

Secure communication protocols are fundamental building blocks of the current digital economy. *Transport Layer Security* (TLS) alone is responsible for protecting most economic transactions done using the web, with a value so high that it is hard to estimate.

These protocols allow entities to exchange messages or data over a secure channel in the Internet. A secure communication channel has three properties: *authenticity*, *confidentiality*, and *integrity*. Regarding authenticity, in an authenticated channel no one can impersonate the sender. Regarding confidentiality, in a confidential channel only the receiver of the message is able to read that message. Regarding integrity, the messages can not be modified without the receiver detecting it.

Several secure communication channel protocols exist nowadays, with different purposes but with the same goal of securing communication. TLS is a secure communication protocol widely used. 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.

Other two secure communication 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 application layer [18]. It is an extension of the IP protocol that contains two subprotocols: AH and ESP. *Secure Shell* (SSH) is an application-layer protocol used for secure remote login and other secure network services over an insecure network [36].

Such a secure communication protocol becomes insecure when a vulnerability is discovered. Vulnerabilities may concern the protocol's specification, the cryptographic mechanisms used, or specific implementations of the protocol. Many vulnerabilities have been discovered in SSL/TLS originating new versions of the protocol with new security features such as deprecating cryptographic mechanisms or enforcing security measures. Concrete implementations of SSL/TLS have been also found vulnerable due to implementation bugs, causing security breaches and affecting devices worldwide.

vtTLS is a protocol that provides *vulnerability-tolerant communication channels*. These channels are characterized by not relying on individual cryptographic mechanisms, so that if one is found vulnerable (or possibly a few of them) the channels remain secure. The idea is to leverage *diversity* and *redundancy* of cryptographic mechanisms and keys, i.e., the use respectively of different and more than one set of mechanisms/keys. This use of diversity and redundancy is inspired in previous works on computer immunology [12], diversity in security [21, 14], and moving-target defenses [6].

In the context of vtTLS, diversity and redundancy con-

DOI: 10.1145/1235

sist in using two or more different mechanisms with the same objective. For example, SHA-1 and SHA-3 are both 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, the *diversity factor* ($k > 1$). This parameter 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 k vulnerabilities exist. As vulnerabilities and, more importantly, zero-day vulnerabilities that cannot be removed as they are unknown [3], 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. It also presents and experimental evaluation of the protocol and shows that it has an acceptable overhead in relation to the TLS implementation in which our prototype is based, OpenSSL v1.0.2g [33].

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. 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 in Security

The term *diversity* is used to describe multi-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 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 no versions [21]. They also state that the key to achieve effective diversity is that the dependence between the different programs needs to be 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 [10]. There are two types of moving-target defenses: reactive and proactive [6]. 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 defenses are as good as their ability to make an unpredictable change to the system.

Forrest *et al.* studied analogies between immunology and computer systems [12]. They claimed that introducing diversity in a deliberate fashion into computer systems can make them more robust to replicated attacks [13].

Earlier Avižienis and Chen introduced N-version programming (NVP) [2]. NVP is defined as the independent generation of N $\geq$ 2 functionally equivalent programs from the same initial specification called *versions*. 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 programming teams, or 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 purpose [14]. Homescu *et al.* use profile-guided optimization for automated software diversity generation [16]. TRR dynamically and randomly relocates program's parts to provide different versions [35]. Proactive obfuscation aims to generate replicas with different vulnerabilities [26]

2.2 SSL/TLS Protocol

This section presents the SSL/TLS, as this is the protocol in which vtTLS is based. We start by presenting its two main sub-protocols: the TLS Handshake Protocol and the TLS Record Protocol.

2.2.1 TLS Handshake Protocol

The TLS 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:

1. The client sends a ClientHello message to the server. This message is sent when the client wants to connect to 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 functionality from the server.

2. The server responds with a ServerHello message. The 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 using 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.

6. The server sends a ServerHelloDone message to conclude its first sequence of messages.

7. After receiving the ServerHelloDone message, the client sends its certificate to the server, if requested. 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 sends a ChangeCipherSpec message to the server. This 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.

11. The client sends its 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.

At this point, the session is established. Client and server can now exchange application-layer data through the secure communication channel.

2.2.2 TLS Record Protocol

The TLS Record protocol is the sub-protocol which processes the messages to sent and received after the handshake, i.e., in normal operation.

Regarding an outgoing message, the first operation performed by the TLS Record protocol is fragmentation. The message is divided into blocks. 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. Although there is always an active compression algorithm, the default one is `CompressionMethod.null`. `CompressionMethod.null` is an operation that does nothing, i.e., no fields are altered.

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. The fragment's length must be $2^{14} + 2048$ bytes or less. When using block ciphers, it is also added 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 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.

2.2.3 TLS Vulnerabilities

To show the relevance of our work, we discuss some of the vulnerabilities discovered in TLS in the past. TLS vulnerabilities can be classified in two types: specification vulnerabilities and implementation vulnerabilities. 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 some of the implementations of SSL/TLS, such as OpenSSL. The Internet Engineering Task Force (IETF) released an RFC in February 2015 summaryzing known TLS attacks and vulnerabilities [28]. This section presents some of the most recent.

The most recent attack against a *specification vulnerability* is Logjam, which was discovered in May 2015 [1]. The Logjam 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 export cryptography. This man-in-the-middle attack trades 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. From the client's point-of-view, the server chose a seemingly normal DHE option and proceeds to compute its secret also with weak Diffie-Hellman 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.

Regarding *implementation vulnerabilities*, *Heartbleed*, discovered in 2014, is one of the most recent. Heartbleed was a security vulnerability 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, i.e., to read data from the memory of the victim [5].

2.3 Vulnerabilities in Cryptographic Schemes

To further motivate the relevance of our work, this section presents vulnerabilities in cryptographic mechanisms, specifically in some of those used by the TLS protocol.

2.3.1 Public-key Cryptography

RSA's security is based on two problems: factorization of large integers and the RSA problem [23]. RSA can be considered to be broken when those problems are solved within 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 feasible time. Factorizing a large integer is very different from breaking RSA, which is still secure. 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.

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 available.

2.3.2 Symmetric Encryption

The Advanced Encryption Standard (AES), originally 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 [4]. 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 [11]. Although it shows some advance in breaking AES, AES with a key of 128 bits has 10 rounds.

2.3.3 Hash Functions

The main applications for hash functions 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 [22]. 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.

The Secure Hash Algorithm 1 (SHA-1) is a cryptographic hash function which produces a 160-bit message digest. Although there is no public knowledge of collisions for SHA-1, it is no longer recommended [9]. Other attacks have been successful against SHA-1. Stevens *et al.* presented a freestart collision attack for SHA-1's internal compression function [32]. 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). 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} [34]. 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 $2^{65.3}$ [31].

3. VULNERABILITY-TOLERANT TLS

vtTLS is a protocol for diverse and redundant 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 in 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 be used in the communication.

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, our 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. Diversity and redundancy will be introduced in the following communication between client and server. 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. The names of the messages are identical in order to provide easier migration and transition from TLS. Using this simplification, the reader familiarized with

TLS can more easily understand vtTLS.

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

The first message to be sent is ClientHello. Its purpose is to inform the server that the client wants to establish a secure channel for communication. The content of this message consists in the client's protocol version, a Random structure (analogous to TLS 1.2) containing the current time and a 28-byte pseudo-randomly generated number, the session identifier, a list of the client's cipher suites and a list of the client's compression methods, if compression is to be used.

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. It also sends the compression method to use, if compression is enabled.

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 generates 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 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 TLS 1.2 [8].

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 ($0 \leq j \leq k$).

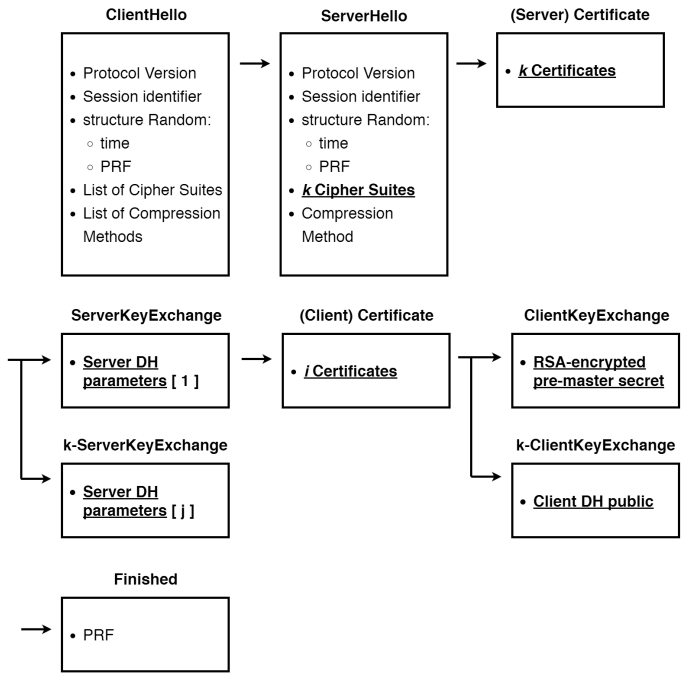

Figure 1: vtTLS Handshake messages using diversity factor k. The places where diversity and redundancy are introduced are marked in bold and underlined.

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 may need to verify the client's i certificates. If they have signing capabilities (i.e., they do not contain fixed Diffie-Hellman parameters), 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 to each party 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 [7]. Moreover, not all cryptographic mechanisms can be used together in the context of TLS 1.2 and other security protocols.

Diversity can be assessed using different metrics. For hash functions, example metrics are origin, year, digest size, structure, rounds and known weaknesses (collisions, second preimage and preimage). After comparing several hash functions using the metrics stated above, Carvalho concluded that the best three combinations are the following:

- SHA-1 + SHA-3: This combination is not possible in vtTLS as SHA-1 is not recommended and TLS 1.2 does not support SHA-3;
- SHA-1 + Whirlpool: This combination is not possible in vtTLS as SHA-1 is not recommended and TLS 1.2 does not support Whirlpool;
- SHA-2 + SHA-3: This combination is 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 (MAC-then-Encrypt mode using a SHA-2 variant) 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.

Regarding public-key functions, the metrics proposed include origin, year, mathematical hard problems, perfect forward secrecy, semantic security and known attacks. After comparing several public-key encryption mechanisms, using the metrics stated above, Carvalho concluded that the best four combinations are:

- DSA + RSA: This combination is 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: This combination is not possible as TLS 1.2 does not support Rabin-Williams;
- RSA + ECDH: This combination is possible as TLS 1.2 supports both functions for *key exchange*;
- RSA + ECDSA: This combination is 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 [15]. 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 did not present any conclusions regarding symmetric-key encryption. Therefore, considering the metrics – origin, year, and semantic security – employed for public-key encryption functions, and considering an additional metric – the mode of operation – we obtained combinations of diverse symmetric-key encryption functions, all possible:

- AES256-GCM + CAMELLIA128-CBC;
- AES256-CBC + CAMELLIA128-GCM;
- AES128-GCM + CAMELLIA256-CBC;
- AES128-CBC + CAMELLIA256-GCM.

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: This combination is 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 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, maximum diversity is achieved using a diversity factor $k = 2$. The least diversified part of the communication is the MAC, due to the fact that TLS 1.2 does not support SHA-3 for now.

4. IMPLEMENTATION

vtTLS' implementation is a modified version of OpenSSL v1.0.2g.[1] Implementing a secure communication channel from scratch would be a bad option as it might lead to the creation of vulnerabilities; existing software such as OpenSSL has the advantaged of being extensively debugged, although serious vulnerabilities like Heartbleed still appear from time to time. Furthermore, creating a new secure communication channel, 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 and keeping the same API. Nevertheless, OpenSSL is a huge code base (currently 438,841 lines of code) and modifying it so support diversity was quite a challenge.

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

```
const char* SSL_get_n_cipher(short n, const SSL* s);
const SSL_CIPHER* SSL_get_current_n_cipher(short n, const SSL* s);
int SSL_CTX_use_n_certificate(short n, SSL_CTX* ctx, X509* x);
int SSL_CTX_use_n_certificate_file(short n, SSL_CTX* ctx, const char* file, int type);
int SSL_CTX_use_n_PrivateKey(short n, SSL_CTX* ctx, EVP_PKEY* pkey);
int SSL_CTX_use_n_PrivateKey_file(short n, SSL_CTX* ctx, const char* file, int type);
int SSL_CTX_check_n_private_key(short n, const SSL_CTX* ctx);
X509* SSL_get_n_peer_certificate(short n, const SSL* s);
```

Figure 2: vtTLS API: additional functions in relation to the OpenSSL API.

Although being based on OpenSSL, vtTLS is not compatible with OpenSSL. Due to its diversity and redundancy features, vtTLS can not connect to a OpenSSL server or client. Moreover, the current vtTLS prototype does not support all OpenSSL cipher suites, e.g., PSK and SRP.

vtTLS adds a few functions to the OpenSSL API. These functions are represented in Figure 2. The meaning of the functions is pretty straightforward. They allow defining additional certificates, keys, cipher functions, etc. The parameter `n` should be set to the number of the certificate, key, etc. being added. For example, with $k = 2$ the parameter `n` takes only value 2 as we have to add just the second of each. For $k = 3$ every function has to be called twice, with parameter `n` set to 2 and 3.

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

The signing and encryption ordering is also important for vtTLS. Figure 3 shows the ordering for one cipher and one MAC in the OpenSSL implementation.

The approach taken was the following, ordered from first to last:

1. Apply the first MAC to the plaintext message;
2. Encrypt the first message and its MAC with the first encryption function;
3. Apply the second MAC to the first ciphertext;
4. Encrypt the first ciphertext and its MAC with the second encryption function.

We opted to maintain the signing prior to the encryption. Using this approach, both message and MACs are encrypted with both ciphers. Figure 4 shows the final ordering of vtTLS communication.

In relation to the *Record Protocol*, signing and encrypting k times has a cost in terms of message size. Figures 3 and 4 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 3). For vtTLS, the additional size of the message is $eivlen\_sec + first\_len + padding\_sec + mac\_size\_sec$, where *eivlen_sec* is the size of the IV associated with the second cipher and *mac_size_sec* the size of the second MAC.

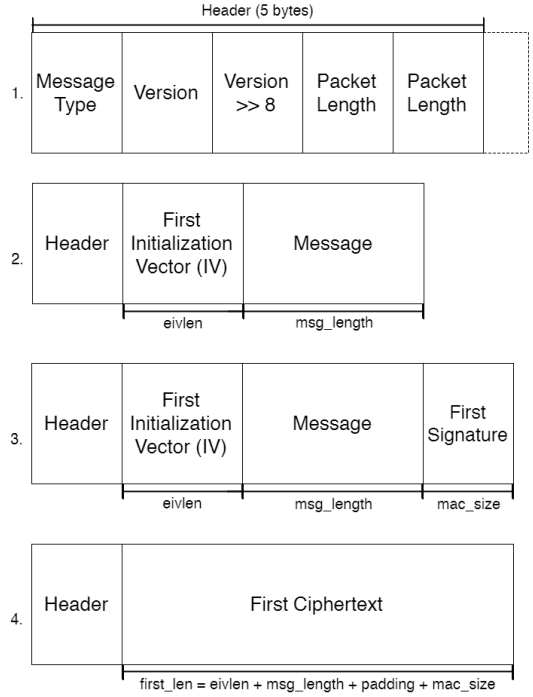

Figure 3: First four steps regarding the ordering of the encryption and signing of vtTLS using a diversity factor $k = 2$.

In the best case, the number of packets is the same for OpenSSL and vtTLS. In the worst case, one additional packet may be send if the encryption function requires a 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. EXPERIMENTAL EVALUATION

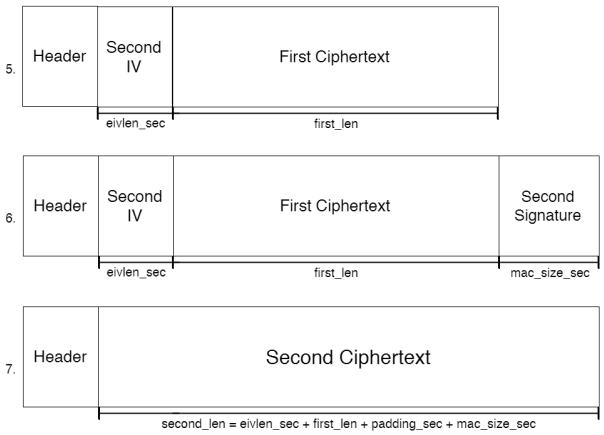

Figure 4: Remaining three steps regarding the ordering of the encryption and signing of vtTLS using a diversity factor $k = 2$. Here is represented the second signing and the second encryption, employing diversity and redundancy in the communication.

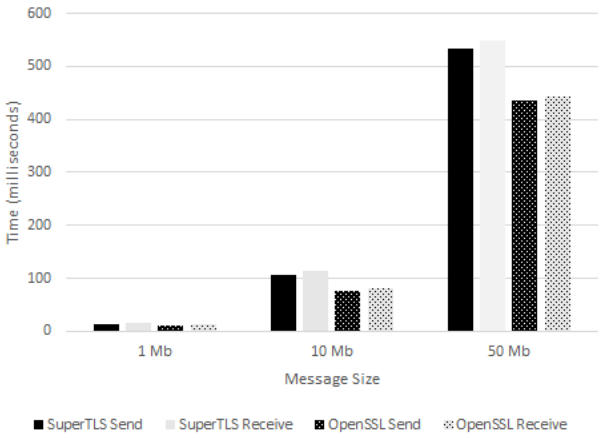

Figure 5: Comparison between the time it takes to send and receive a message using vtTLS and OpenSSL 1.0.2g.

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

Implementing 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, such as OpenSSL, 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).

In order to perform this 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 locations in order to maintain the evaluation valid, exact and precise.

5.1 Performance

In order to evaluate vtTLS' performance, 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: `TLS_RSA_WITH_AES_256_GCM_SHA384` and `TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384`. The suite used with OpenSSL was the second.

5.1.1 Handshake

To evaluate the performance of the handshake, we executed 100 times the Handhake Protocol of both vtTLS and OpenSSL. In average, the vtTLS' handshake took 3.909 milliseconds to conclude and the OpenSSL's handshake 2.345 milliseconds. Therefore, the vtTLS's Handshake takes 66.7% longer than OpenSSL's Handshake. The vtTLS' Handshake is only 1.67 slower than the OpenSSL's Handshake, which is better than the worst case, 2 times.

5.2 Data Communication

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 using vtTLS, and other 100 messages sent and received using OpenSSL.

Figure 5 shows the comparison between the time it takes to send and receive a message using vtTLS and OpenSSL/TLS. The measurements concern the time each channel needs to perform the local operations 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`.

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, in average 534.55 ms to be sent with vtTLS. Using OpenSSL, the same message takes 435.01 ms to be sent. The overhead generated by using diverse encryption and signing mechanisms exists, as expected, but it is much smaller than the worst case.

In Section 4, we do an analysis of the expected message size increase. 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. Using 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. Although, 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 converts into a message of 1,025,856 bytes. Concluding, sending 1 MB through a vtTLS channel costs an additional 3,198 bytes than using a OpenSSL channel.

5.3 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. Although certificates are not expensive, they represent a cost. Using a diversity factor $k = 2$, the cost associated with certificates duplicates. Nevertheless, we believe that the additional cost of having two certificates instead of one is compensated by the increase in security and vulnerability tolerance provided by vtTLS. Regarding management, there is the need to manage two certificates instead of one. This does not represent a substantial increase in management effort.

6. CONCLUSIONS

vtTLS is a diverse and redundant vulnerability-tolerant secure communication protocol designed for communication between clouds. 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 seems to provide an interesting trade-off for a set of critical applications.

7. 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*, October 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] 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.

[4] 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.

[5] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler. Heartbleed 101. *IEEE Security Privacy*, 12(4):63–67, July 2014.

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

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

[8] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008. http://www.rfc-editor.org/rfc/rfc5246.txt.

[9] ENISA. Algorithms, key size and parameters report – 2014. nov 2014.

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

[11] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting. Improved cryptanalysis of Rijndael. In G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier, editors, *Proceedings of Fast Software Encryption*, volume LNCS 1978, pages 213–230. Springer, 2001.

[12] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, Oct. 1997.

[13] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*. IEEE Computer Society, 1997.

[14] 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.

[15] 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. ACM, 2002.

[16] 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.

[17] S. Kanno and M. Kanda. Addition of the Camellia Cipher Suites to Transport Layer Security (TLS). RFC 6367, RFC Editor, September 2011.

[18] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, RFC Editor, December 2005. http://www.rfc-editor.org/rfc/rfc4301.txt.

[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, pages 321–383. CRC Press, Inc., 1996.

[23] A. Menezes, P. van Oorschot, and S. Vanstone. Public-Key Encryption. In *Handbook of Applied Cryptography*, chapter 8. CRC Press, Inc., 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 Trans. Comput. Syst.*, 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, RFC Editor, February 2012. http://www.rfc-editor.org/rfc/rfc6520.txt.

[28] Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). RFC 7457, RFC Editor, February 2015. http://www.rfc-editor.org/rfc/rfc7457.txt.

[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, Apr. 2010.

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

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

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

[34] 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.

[35] 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.

[36] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251, RFC Editor, January 2006. http://www.rfc-editor.org/rfc/rfc4251.txt.