SafeCloud

Final version of the private communication middleware

Deliverable 1.3

Project reference no. 653884

February 2018

European Commission

Horizon 2020
European Union funding
for Research & Innovation

Document information

Scheduled delivery	01.03.2018
Actual delivery	01.03.2018
Version	1.0
Responsible partner	INESC-ID

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
08.01.2018	M. Pardal	Draft	0.0	Initial TOC
15.01.2018	S. Totakura	Draft	0.1	sKnock revision and integration
19.01.2018	H. Niedermayer	Draft	0.2	Additional protections
27.01.2018	M. Pardal	Draft	0.3	vTTLS revision and integration
30.01.2018	M. Pardal	Draft	0.4	Chapter structure review
31.01.2018	H. Niedermayer	Draft	0.5	Ready for review
27.02.2018	M. Pardal	Draft	0.6	Incorporate reviews
28.02.2018	M. Pardal	Final	1.0	Final version

Contributors

M. Pardal (INESC-ID)
M. Correia (INESC-ID)
A. Joaquim (INESC-ID)
K. Balu (INESC-ID)
D. Raposo (INESC-ID)
I. Costa (INESC-ID)
R. Moura (INESC-ID)
S. H. Totakura (TUM)
H. Niedermayer (TUM)

Internal reviewers

J. Paulo (INESC TEC)
H. Mercier (UniNE)

S. Schmerler (Cloud & Heat)

Acknowledgments

This project is partially funded by the European Commission Horizon 2020 work programme under grant agreement no. 653884

More information

Additional information and public deliverables of SafeCloud can be found at:
`http://www.safecloud-project.eu/`

Contents

List of Figures

List of Tables

Executive Summary

This document gives an overview of the three solutions in the Secure Communications layer of the SafeCloud project.

The SafeCloud middleware components are open-source and available in the following locations:

- vTTLS: `https://github.com/safecloud-project/vtTLS`
- sKnock: `https://github.com/safecloud-project/sKnock`
- Darshana: `https://github.com/safecloud-project/darshana`
- Machete: `https://github.com/safecloud-project/MACHETE`

1 Introduction

Data communication is one of the most significant source of concerns about privacy and confidentiality in the Internet. The objective of work package WP1 is to provide middleware services to improve the privacy and security of cloud communications in the SafeCloud architecture. The SafeCloud secure communication has three solutions as shown in Figure 1.1

Secure communication		
SC1 **Vulnerability-tolerant channels**	**SC2** **Protected channels**	**SC3** **Route-aware channels**

Figure 1.1: SafeCloud Secure Communication Solutions.

The purpose of these communication services is to provide the same properties as *secure channels* - confidentiality, integrity, and authenticity - plus availability - but assuming powerful adversaries that may be able to break some of the assumptions that make existing channels secure. A specific example of such a broken assumption is to consider that the Diffie Hellman key exchange is secure, proved wrong recently by the Logjam attack [ABD+15].

Deliverable D1.1 characterized the threats, communication services, and presented the overall architecture of the SafeCloud middleware: it is aimed at machine-to-cloud and cloud-to-cloud communication; it provides only unicast communication between two endpoints (e.g. user terminals, servers, etc). The communication model is connection-oriented, similarly to protocols like TCP or SSL. The middleware is implemented at the application layer of the OSI model and of the Internet protocol stack, so that the approach is applicable to as many use cases as possible, as solutions at the lower layers would require an unfeasible modification of the equipment of Internet service providers.

Deliverable D1.2 presented an initial version of the secure communication middleware components: vulnerability-tolerant channels (vTTLS), protected service provisioning (sKnock), route monitoring (Darshana), and multi-path communication (MACHETE).

The present deliverable, D1.3, presents the final version of the middleware. It is organized as follows:

- Chapter 2 presents the business requirements that motivated the middleware components.
- Chapter 3 presents vTTLS, a *vulnerability-tolerant communication channel* based on diversity and redundancy of the cryptographic algorithms.
- Chapter 4 presents the work on protected service provisioning. This includes considerations about certificates and safer authentication and sKnock, SafeCloud's approach to *port-knocking*. This concept hides remote services behind a firewall which allows access to the listening ports of the services only after the client has successfully authenticated to the firewall.

- Chapter 5 presents Darshana, a *network route monitoring* solution that detects route hijacking caused by Border Gateway Protocol (BGP) or other network tampering. The solution uses active probing techniques that enable detection in near real-time.

- Chapter 6 presents Machete, an application-layer *multi-path communication* mechanism that provides additional confidentiality by splitting data streams in different physical paths.

- Chapter 7 describes how added-value can be derived based on the integration of two or more SafeCloud middleware components. The vTTLS and sKnock integration is presented in detail.

- Finally, the document concludes in Chapter 8.

2 Requirements

This chapter provides our analysis of the business requirements made in WP1. We addressed the need for communication security, the limitations of the industry-standard solutions, and how the SafeCloud Secure Communication solutions address some of the limitations.

2.1 Perceived need of communication security

Confidentiality, privacy and integrity are widely-acknowledged requirements for communication over the Internet and with clouds. These requirements became more compelling with the Snowden revelations and news on generalized eavesdropping of communications.

There are many reports, pieces of news, and other information that support these assertions. A recent piece of news at CNET pointed out that “In the three years since Snowden’s initial leak, Apple, Google, Microsoft, Facebook and Yahoo have become some of the biggest advocates of consumer privacy. They’ve beefed up encryption and other safeguards in their products and services” [Hau16]. According to the same source, these companies took such measures “to protect business”. This trend is also shown by the coalition of 40 large tech companies around Apple to block the FBI’s access to data stored in iPhones [Che16].

Not necessarily related to these revelations, Vodafone, the second largest mobile phone operator worldwide (after China Mobile) also considers that “Customer information is one of the greatest assets we are entrusted with and must be protected appropriately” [Vod14]. Among their “Key Principles on Information Security”, Vodafone lists “Confidentiality: Customer information must not be disclosed to, or accessed by, unauthorised people” and “Integrity: Customer information and software must be accurate, complete and authentic so that it can be relied upon.”

Microsoft also lists privacy/confidentiality in the list of “What customers want from cloud providers” in a recent report on Windows Azure, one of the largest cloud services worldwide [Mic15]. This list contains 5 items, two of them directly related with these properties: Secure our data; Keep our data private; Give us control; Promote transparency; Maintain compliance. The report confirms that “Since the revelations of widespread surveillance by the US government in 2013, privacy concerns have become more accentuated, and the cloud has come under greater scrutiny as a result.” In relation to data sent over the network it says that “For data in transit, customers can enable encryption for traffic between their own VMs and end users” and that “Azure uses industry-standard transport protocols between devices and Microsoft datacenters and within datacenters themselves.”

These concerns exist not only in companies but also among the general public. The Pew Research Center study made in 2014 found out that there are major concerns on surveillance by government and businesses, which justifies both the need for secure communications but also storage and processing (WP2 and WP3) [Pew14]. The report points out specifically that there is little confidence in the security (privacy/confidentiality) of common communication channels.

On a more official level, the ITU-T, part of the International Telecommunication Union and a major standards committee in communications, published in 2012 a report specifically focused on privacy in cloud computing and its communications [IT12]. Also the Cloud Security Alliance, arguably the most important think tank in cloud security, includes "Encryption and Key Management" among its Critical Areas of Focus [Clo11]. Although the document focuses more on encryption of data inside the cloud, it makes clear that a "secure transfer channel (i.e. TLS)" is essential for communication protection. The American Federal Communications Commission published a set of cyber-security recommendations for small companies [Fed16]. It provides counsel such as "Secure internal network and cloud services" and "Encryption should be employed to protect any data that your company considers sensitive (...) on disk as well as in transit". The National Telecommunications and Information Administration from the United States Department of Commerce also concluded that lack of trust in Internet privacy and security may be a major deterrent for online business [Gol16].

In summary, there is a large consensus on the need for communication security and privacy, in general in the Internet but also in particular with clouds. Privacy and confidentiality are the major concerns, but integrity is also often mentioned as a problem.

2.2 Limitations of industry-standard solutions

The industry-standard solution to the problems mentioned in the previous section is the use of protocols that provide secure channels, often designated informally as "encryption" [Clo11, Fed16, Hau16, Mic15]. The most adopted protocol is SSL/TLS, which has several variants. This protocol provides transport layer channels, and is widely-adopted for protecting application layer communication in combination with the HTTP protocol. This bundle of SSL/TLS and HTTP is usually denominated HTTPS.

SSL/TLS provides to some extent the properties required by companies, governments, and the general public that we mentioned in the previous section: privacy, confidentiality, and integrity of communications. However, they only provide these properties as long as no vulnerabilities exist in its components: cryptographic hash functions, encryption algorithms, message authentication code schemes, padding schemes, authentication protocols, client and server-side software, etc. In reality, channels based on SSL/TLS are sometimes insecure for several reasons:

1. A vulnerability appears in one of these components;
2. An old vulnerability in one of these components is not fixed;
3. There is an unknown (or 0-day) vulnerability in one of these components;
4. There is a vulnerability that seems to be impossible to exploit, but that can be exploited by a strong adversary (e.g., a nation state).

In deliverable D1.1 we already provided extensive references to scientific work that justifies the existence of such vulnerabilities. Nevertheless, we provide some additional evidence based on very recent research.

In relation to point (1), there are a few vulnerabilities recently published. A specific example is a vulnerability in the ECDSA signature algorithm implementation in OpenSSL [FWC16]. Another example is a collision attack on HTTP over TLS [BL16]. Finally, another paper in the same conference provides a mechanisms to do fuzz testing of TLS libraries, with the objective of finding implementation vulnerabilities in the source code [Som16].

In relation to point (2), another recent paper shows that vulnerabilities in secure channels take a long time to be fixed [ADHP16b].

In relation to (3), a paper with a few more years presents a study about the duration of 0-day attack campaigns against generic vulnerabilities (not necessarily secure channels), based on extensive data from Symantec [BD12]. It concludes that the duration of such campaigns varies between 19 days and 30 months.

Finally, in relation to (4), a very good example appeared in 2015: *Logjam* [ABD+15]. That vulnerability allows downgrading the strength of Diffie-Hellman key-exchange of TLS, then get access to a decryption key and eavesdrop on communication. The attack however requires the precomputation of a specific group, which in the case of 1024-bit groups can be done by nation state adversaries.

2.3 Addressing the requirements in SafeCloud

For the attacker to break the confidentiality, privacy or integrity of an SSL/TLS channel he must:

- (i) know about the existence of a vulnerability in the channel; and
- (ii) have access to the endpoints or
- (iii) have access to the communication.

SafeCloud's middleware provides a set of mechanisms that make it harder for the attacker to get (i), (ii), and (iii). Specifically, the middleware services do the following:

- Vulnerability-tolerant channels (task T1.2) – targets (i) by requiring more than one vulnerability for attacks to be effective;
- Protected service provisioning (task T1.3) – targets (ii) by hiding the endpoints and enhancing authentication;
- Route monitoring (task T1.4) and Multi-path communication (task T1.5) – target (iii) by detecting if the attacker deviated the traffic to have access to it; and by splitting the traffic, thus making it more difficult for the attacker to have access to data.

The following chapters present the SafeCloud middleware components along with details about the security and performance trade-offs that they entail.

3 Vulnerability-Tolerant Channels

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.

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 [KS05]. 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 [YL06].

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 [ADHP16a]. 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.

vtTLS is a protocol, based on TLS, that provides *vulnerability-tolerant communication channels*. vtTLS stands for *Vulnerability-Tolerant Transport Layer Security*. 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 [VNC03], diversity in security [LS04, GBG$^+$11] and moving-target defenses [CF14].

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 [BD12], 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 chapter 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 [VMC02].

3.1 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.

3.1.1 Diversity

The term *diversity* is used to describe multiple version software in which redundant versions are deliberately created and made different between themselves [LS04]. 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 [LS04]. 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 [ENTK11]. There are two types of moving-target defenses: proactive and reactive [CF14]. 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 [SBC$^+$10]. 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) [AC77]. NVP is defined as the independent generation of N ≥ 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 [LHBF14]. Garcia *et al.* show that there is enough diversity among operating systems for several practical purposes [GBG$^+$11]. Homescu *et al.* use profile-guided optimization for automated software diversity generation [HNL$^+$13]. Transparent Runtime Randomization (TRR) dynamically and randomly relocates parts of the program to provide different versions [XKI03]. Proactive obfuscation aims to generate replicas with different vulnerabilities [RS10].

3.1.2 SSL/TLS Protocol

Here we present 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.

3.1.2.1 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.

3.1.2.2 TLS Record Protocol

The TLS 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 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.

3.1.2.3 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 [SHSA15].

An example attack that exploits a specification vulnerability is *Logjam* [ABD+15]. 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. One aspect worth noticing is that this attack will only succeed if the server does not refuse to accept DHE_EXPORT mode. The solution for this vulnerability is simple and has already been implemented: browsers simply deny the access to servers using weak Diffie-Hellman cipher suites, such as DHE EXPORT, although TLS still allows it.

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 [STW12]. The Heartbleed vulnerability allowed an attacker to perform a buffer over-read, reading up to 64 KB from the memory of the victim [CDFW14].

3.1.3 Vulnerabilities in Cryptographic Schemes

We present vulnerabilities in cryptographic mechanisms, specifically in some of those used by the TLS protocol. Not every mechanism supported by TLS is secure. Several of these mechanisms have vulnerabilities which can make the communication insecure, if exploited.

3.1.3.1 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 [MvOV96]. 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 [KAF$^+$10]. 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 [ENI14].

Shor designed a quantum computing algorithm to factorize integers in polynomial time [Sho95]. However, it requires a quantum computer able to run it, which is still not publicly available.

3.1.3.2 Symmetric Encryption

The Advanced Encryption Standard (AES), originally called Rijndael, is the current American standard for symmetric encryption [RD01]. 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 [BKR11]. 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 [FKL+01]. Although it shows some advance in breaking AES, AES with a key of 128 bits has 10 rounds.

3.1.3.3 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 [MvOV96]. 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 [ENI14], although the first collision was discovered only recently [SBK+17]. There have been some previous attacks against SHA-1. Stevens *et al.* presented a freestart collision attack for SHA-1's internal compression function [SKP15]. Taking into consideration the Damgard-Merkle [Mer79] 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} [WYY05]. 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 $2^{65.3}$ [Ste12].

3.2 Vulnerability-Tolerant TLS

vtTLS is a new protocol that provides vulnerability-tolerant secure communication channels. It aims at increasing security by 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, 3.1 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. 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.

3.2.1 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, 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 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

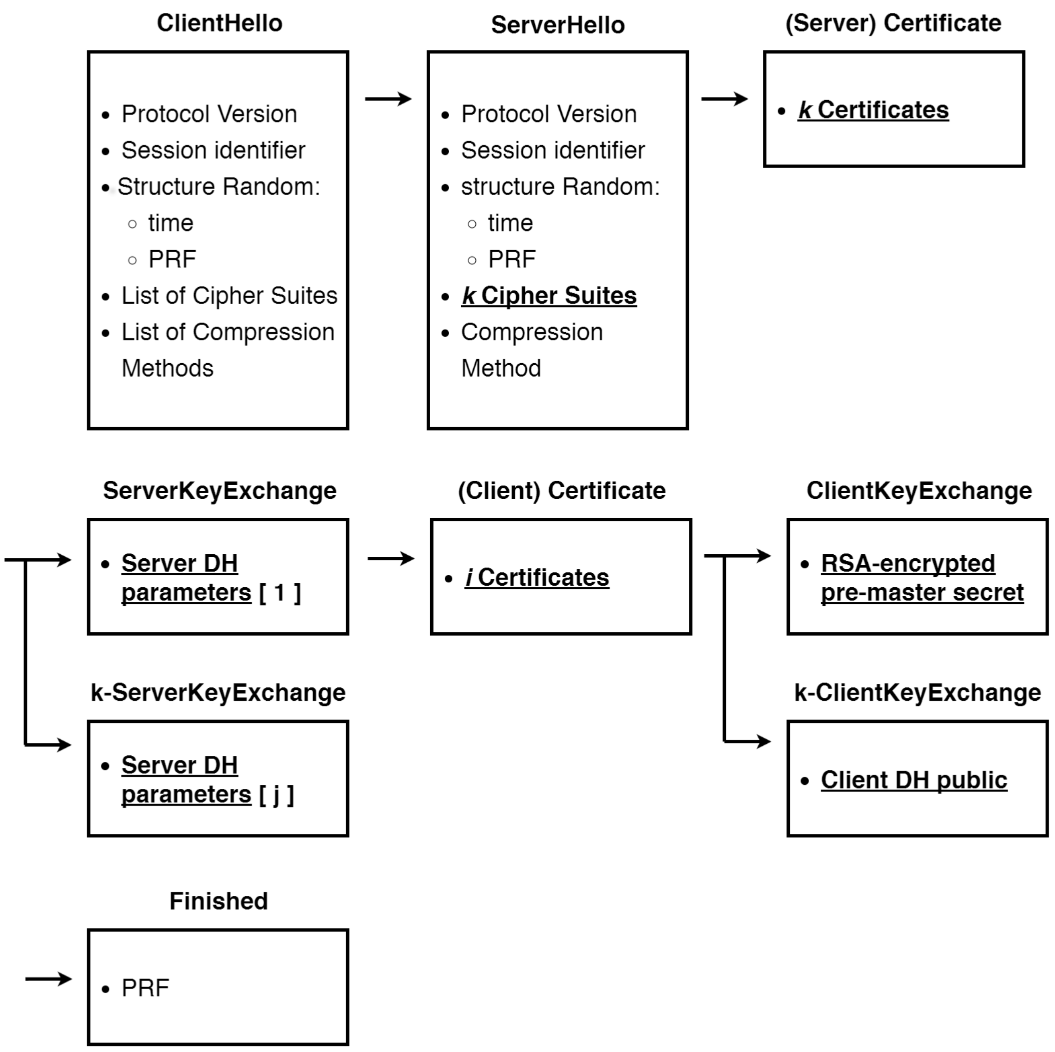


Figure 3.1: vTTLS handshake messages using diversity factor k. The points where diversity and redundancy are introduced are marked in bold and underlined.

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 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 [DR08].

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.

3.2.2 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 [Car14]. 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.

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 [GGCS02]. 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 [KK11] 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, 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.

3.3 Implementation

The 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 compatible with it due to its diversity and redundancy features. It is noteworthy that OpenSSL is a very large code base (438,841 lines of code in version 1.0.2g) so modifying it to support diversity was a considerable engineering challenge.

vTTLS adds a few functions to the OpenSSL API. These functions are represented in Figure

[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 3.2: vTTLS API: additional functions in relation to the OpenSSL API.

3.2. 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 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 3.3 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 3.4 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.

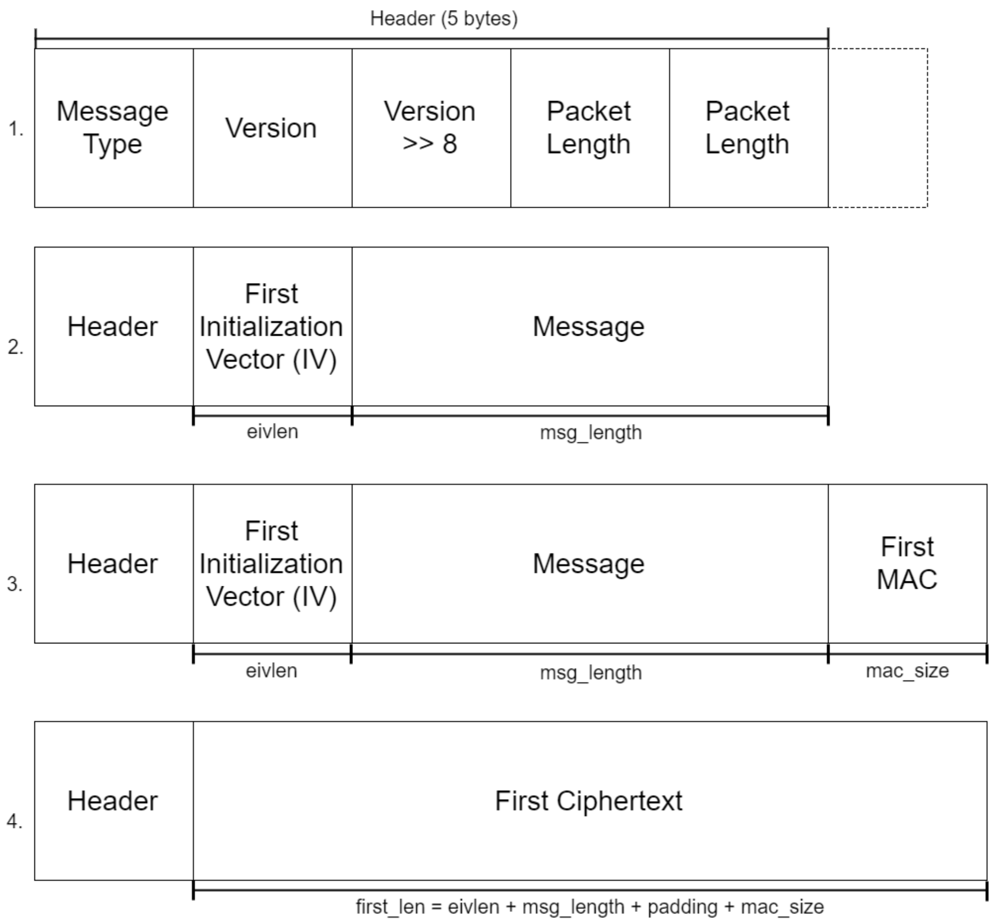


Figure 3.3: First four steps regarding the ordering of the encryption and MAC of vTTLS using a diversity factor $k = 2$.

In relation to the *Record Protocol*, signing and encrypting k times has a cost in terms of message size. Figures 3.3 and 3.4 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 3.3). 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.

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.

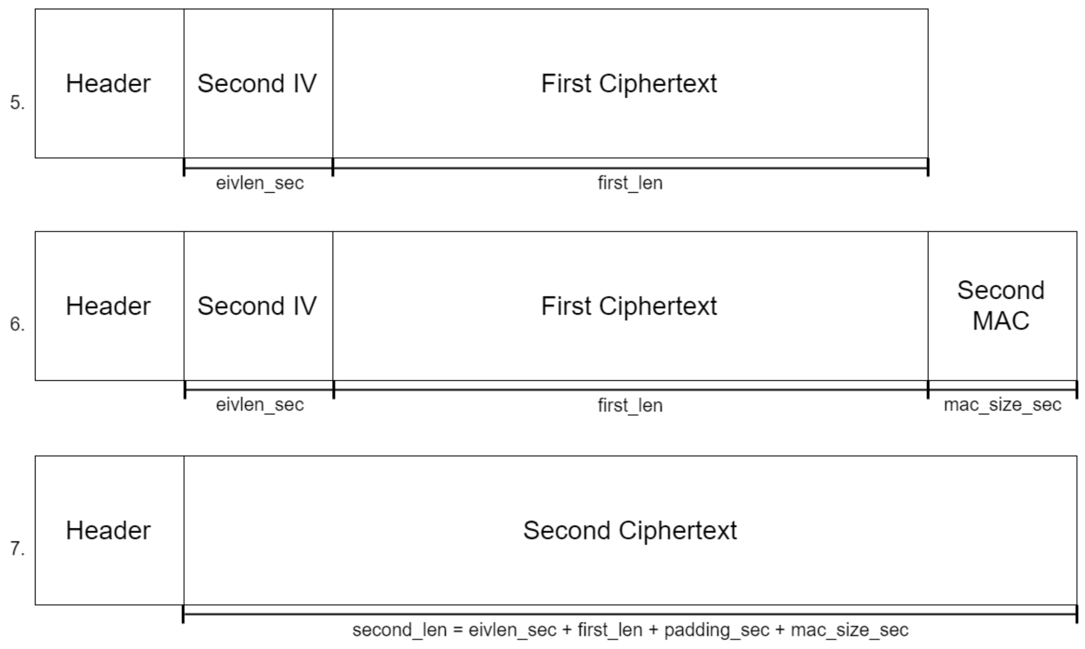


Figure 3.4: Remaining three steps regarding the ordering of the encryption and MAC of vTTLS using a diversity factor $k = 2$. Here is represented the second signing and the second encryption, employing diversity and redundancy in the communication.

3.4 Experimental 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.

3.4.1 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: `TLS_RSA_WITH_AES_256_GCM`

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

Table 3.1: Handshake time comparison

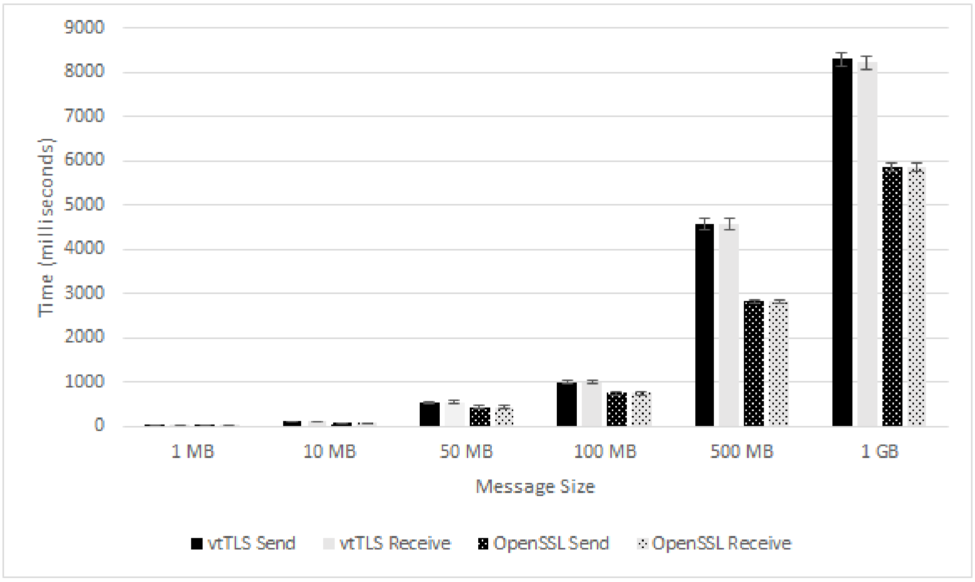


Figure 3.5: Comparison between the time it takes to send and receive a message using vTTLS and OpenSSL.

`_SHA384` and `TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384`. The suite used with OpenSSL was the second.

3.4.1.1 Handshake

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 3.1 provides more details.

3.4.2 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 with vTTLS and 100 messages sent and received with OpenSSL.

Figure 3.5 shows the comparison between the time it takes to send and receive a message with vTTLS and OpenSSL/TLS. Tables 3.2 and 3.3 show more details.

	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 3.2: 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 3.3: Time to receive a message with vTTLS and OpenSSL.

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`. 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 Section 3.3, we did 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. 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

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 3.4: vTTLS and OpenSSL message sizes (in bytes).

Table 3.5: Comparison of time overhead of adding a second layer of protection (going from $k = 1$ to $k = 2$) in vTTLS and in MultiTLS

Message size	vTTLS overhead	MultiTLS overhead
1 MB	9.05%	60.00%
100 MB	32.68%	141.65%
1 GB	40.61%	175.93%

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 3.4 shows all the results obtained and the comparison between the message sizes.

3.5 Comparison with tunneling approach

To further validate the vTTLS approach, we developed an alternative implementation of the same idea, but using *tunneling*[2]. The term tunneling describes a process of encapsulating entire data packets as the payload within other packets, which are handled properly by the network on both endpoints [Sim95].

This alternative approach, that we called MultiTLS, allows adding the multiple layers of protection, but without having to modify the internals of the TLS implementation, which has potential for better code maintenance over time. MultiTLS uses the socat tool, a variant of netcat, that establishes two bidirectional byte streams and transfers data between them. Socat is integrated with OpenSSL to provide security, and with tun/tap virtual network adapters [CK15] to provide tunneling.

To evaluate MultiTLS, we ran performance tests on a setup with 2 virtual machines, client and server, both running Debian 8, in the same Intel Core i7 host computer with 8 GB RAM. We measured the time of receiving messages of different sizes: 1 MB, 100 MB, and 1 GB.

Table 3.5 compares the overhead of going from one layer of protection ($k = 1$) to two layers of protection ($k = 2$) in vtTLS, and the same in MultiTLS. We can see that the overheads in MultiTLS are an order of magnitude higher, in all reported message sizes. The vtTLS overhead is below the 100% threshold in all cases. We can conclude that, even though MultiTLS is more flexible, it comes at a significant higher cost, making vTTLS the best option at providing both multiple protection and good performance.

3.6 Summary

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.

To evaluate the solution, it was compared with 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.

We also compared vtTLS with an alternative implementation, MultiTLS, using tunneling to achieve the same protection, and found the vtTLS has an order of magnitude less overhead.

Overall, considering the additional costs of having an extra certificate, the time increase, and potential management costs, vtTLS provides a good security-performance trade-off for a set of critical applications.

4 Protected Service Provisioning

This component of the SafeCloud secure communication middleware addresses the questions of how deployed services can be protected, and how can clients accessing these services be assured of their identity. In this chapter, we address the deployment of port-knocking protections, a top state-of-the-art service hardening measures; and we address enhanced protection of service management and certificates.

4.1 Port Knocking

Port-knocking is the concept of hiding remote services behind a firewall which drops all incoming connections to the services by default, but allows them only after the client has authenticated to the firewall. This can be seen as an additional layer of security which provides protection from port scans and exploits on the services from unauthorized clients.

There are many implementations of port-knocking as of today and most of them employ authentication based on shared secrets. In these implementations the firewall is configured to authenticate a client based on its corresponding secret. While this approach is simple and efficient for a small number of clients, it quickly becomes unmanageable for a service provider with a large and dynamically changing client base.

Another problem with authentication based on secrets is that it is common for service providers to offer multiple servers to a common pool of clients for reasons of providing redundancy and load-balancing. In such a setup, the servers or the firewalls protecting the services have to synchronise the port-knocking secrets shared with each client. Furthermore, when a service provider uses a cloud provider for service delivery, the secrets have to be configured in the cloud provider's infrastructure which may give away the size of the client base of the service provider to the cloud provider.

To our knowledge there exists no implementation of port-knocking which demonstrates scalability on-par with what is required for a service provider using cloud computing infrastructure. To overcome the scalability barrier, we propose *sKnock* (named after 'scalable Knock'), our approach towards port-knocking using X.509 [ITU05] certificates to address scalability. The motivation behind using certificates is that a certificate's validity can be checked by having the certificate of the signer; while adhering to X.509 allows us to encode authentication information as X.509 extensions and be able to use renowned libraries such as OpenSSL to parse the certificates. This, albeit a lookup in the revocation database which is usually small, aids in improving the scalability of authentication. Moreover, this approach is not subjected to the problem of synchronising the secrets among different servers and mitigates the privacy problem of knowing the size of client base as the firewall now only requires the certificate of the certification authority (CA) to authenticate the clients.

The client component provides a C library which allows applications to integrate port-knocking functionality. The server component integrates with the firewall and dynamically configures it to allow authenticated clients to communicate with the services behind the firewall.

The attacker model is introduced next and we use it to evaluate some of the existing port-knocking approaches and their authentication process in detail to highlight the differences with our approach in 4.1.2. 4.1.3 introduces sKnock describing the authentication protocol and the design decisions we took to overcome common pitfalls. 4.1.4 describes our evaluations of an implementation of sKnock in Python. Finally, we describe the pros and cons of sKnock compared to others port-knocking approaches in 4.3.

4.1.1 Attacker Model

For an attacker interested in attacking the services protected by a port-knocked firewall, he has to either pose as a valid client of the service or defeat the port-knocking protection. For public services, an attacker can easily become a valid client. Therefore, we consider an attacker model where the attacker is interested in attacking the port-knocked firewall.

In this model, we consider the following capabilities to model different types of attackers:

A1 Record any number of IP packets between a given source and destination and replay them from own IP address.

A2 Modify, and suppress any number of IP packets from a given source and destination.

A3 Send IP packets from any IP address and receive packets destined to any IP address.

Capability A1 can be acquired by an attacker by snooping anywhere on the network route between source and destination. Additionally, if the attacker can position himself as a hop anywhere in the route, he gains capability A2. A3 can be acquired by positioning himself in the link connecting the firewall to the Internet.

In addition to this, we also consider a valid client to be an attacker if the client's validity is revoked due to some reason and the client then tries to exploit the port-knocked firewall. Therefore, we assign the following properties:

B1 Attacker knows that the firewall uses port-knocking.

B2 Attacker may have previously port-knocked successfully as a valid client.

Notice that capabilities B1 and B2 are easier to acquire than A1, A2, and A3. B1 is even easier as it is possible to learn the existence of port-knocked firewall if that information is public (if it is advertised by the service provider).

In the rest of the text, we refer to attackers with a combination of these capabilities using the set notation: an attacker with capabilities A1 and B1 is termed as {A1, B1} attacker. If an attacker has a single capability, we will ignore the braces, *i.e.* A1, B1 attackers mean two type of attackers each with a capability, but not both capabilities.

Finally we assume that attackers cannot decrypt encrypted content and cannot forge signatures for arbitrary parties without the knowledge of their keys. This can be ensured in practise by proper usage of cryptography.

4.1.2 Related Work

Port-knocking is historically done by sending packets to different ports in a predefined static sequence. The sequence is kept secret and is shared with authorised clients. The firewall monitors the incoming packets and opens the corresponding port for a remote service for a client if the destination port numbers of a sequence of packets coming from that client match the corresponding predefined sequence. An A1 attacker can observe this sequence and later replay it to defeat port-knocking.

Variants of port-knocking which use multiple packets to convey authenticating information to the firewall are termed under *Hybrid Port-Knocking*. To defend against A1 attackers, the sequence can be made dynamic with the usage of cryptography, *e.g.* by deriving the sequence from a time based one-time password (TOTP)[MMPR11]. However it is vulnerable to {A1, A2} attackers because the attacker can suppress a suspected sequence of packets from reaching the firewall and replay it from his host.

To defend against {A1, A2} attackers the IP address of the client needs to be encoded into the authenticating information in a way that the firewall can retrieve it to open the port in the firewall for that client. If the client's IP address is in cleartext, then the authenticating information should contain a message authentication code (MAC), *e.g.* through HMAC [KBC97], so that the attacker cannot rewrite it with his IP address.

However, including the client IP address in the authentication information causes problems for clients behind NAT. Such clients are required to know their NAT's public IP address to be able to successfully knock the firewall. This may, however, lead to *NAT-Knocking* attacks [MMETC05] as the NAT's IP address is shared by other clients in the network and one of them could be an attacker. Aycock et. al.[AJ$^+$05] proposed an approach based on challenge-response to solve this problem. This approach requires a three-way handshake where the client initiates by sending a request to the firewall. The request contains an identifier and the client's IP address. The firewall then sends a challenge containing the IP address it observed as the request's source IP, the IP address present in the request, and a random nounce, together with a MAC over these fields. The client then responds to this challenge by presenting a MAC over these fields with its preshared secret with the firewall. Since the handshake messages are authenticated with MAC, this approach is immune to {A1, A2} attackers.

A practical problem with hybrid port-knocking variants is that they fail when packets are delivered out-of-order to the firewall. To address this, the authenticating information could be sent as a single packet. This variant of port-knocking is termed as *Single Packet Authorisation* and is first documented to be used in *Doorman* [Krz03]. Doorman authenticates clients based on a HMAC derived from the shared secret, port number to open for the client, username, and a random number. The random number is used to provide protection against A1 attackers as the firewall rejects a request with a number already seen. Since it does not include the client IP address, it is susceptible to {A1, A2} attackers.

Furthermore, there are variants which perform stealthy port-knocking by encoding the authenticating information into seemingly random looking fields of known protocols, *e.g.* the initial sequence number field of TCP, and the source port number. The advantage of these variants is that they make it difficult for an attacker to suspect port-knocking mechanisms

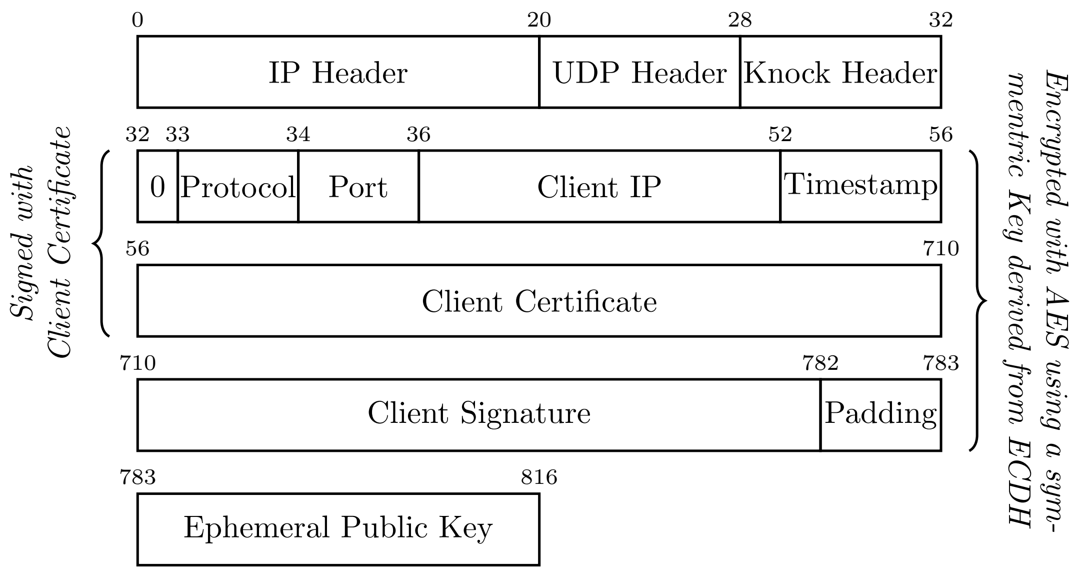


Figure 4.1: Packet format of the sKnock authentication packet starting with a IPv4 header. The fields for *Port* and *Protocol* provide the port number to be opened in the firewall for the given protocol (TCP or UDP). The *Ephemeral Public Key* contains the public key used by the client and is required for the server to determine the AES key using Elliptic Curve Diffie-Hellman (ECDH).

just by observing the traffic. Among these are *SilentKnock* from Vasserman et. al. [VHT09] and *Knock* from Kirsh et. al. [KG14].

SilentKnock encodes authentication token into the TCP header fields of the TCP SYN packet sent by the client. The firewall intercepts this packet from the kernel and extracts the token. The server then verifies the token and opens the corresponding port if the token is valid. The token is generated with keyed MAC with counters to prevent replay attacks from A1 attackers.

Similar to SilentKnock, the stealth property in Knock is achieved encoding the authentication token into the TCP header fields. Additionally, Knock allows for the client and the server to derive a session key which is then used to authenticate application data, thus preventing an {A1, A2, A3} attacker to take over the connection after successful port-knocking.

While the concept of stealthy port-knocking can be applied to any operating system, the current state of implementations for SilentKnock and Knock are limited to the Linux kernel. This poses a deployment barrier for service providers as they have to require their consumers to run complying software setup on their hosts.

4.1.3 sKnock

In all of these approaches presented earlier, the client and the firewall depend on a shared secret to authenticate and gain defence against A1, A2 attackers. In the case of B2 attackers, these approaches require the shared secret to be invalidated at the firewall. This brings in the inconvenience and scalability problems discussed in 4.1.

sKnock addresses the scalability problem by using certificates: each client gets a certificate which it uses to encrypt and sign the authentication information; the firewall requires the CA certificate to authenticate the client. B2 attackers are defended by limiting the certificate validity to an expiry date and having a certificate revocation list to invalidate certificates before their expiration.

Next we describe sKnock's authentication protocol and give a brief description of its implementation in Python.

Protocol

The one-way authentication protocol of sKnock requires the client to send an authentication packet before opening a connection to the remote services behind the firewall. The authentication packet is a UDP packet containing the client's certificate and the port number of the remote service it wants to communicate with on the server. Additionally, it contains the client's IP and timestamp to provide protection against A1, A2 attackers. The format of the packet is shown in Figure 4.1. This information in the packet is encrypted with an ephemeral key which is derived from the server's public key using Elliptic Curve Diffie-Hellman (ECDH). The client's Diffe-Hellman share required for generating the ephemeral key at the server is also included in the packet.

Since we want to keep the overhead of port-knocking low and also reduce the number of packets involved in the authentication to perform well under packet loss, it is important to fit this payload in one UDP packet. The limiting factors here are the network MTU sizes and the size of the client certificate. A common network MTU of 1500 bytes has eliminated the use of RSA and DSA public keys of lengths 2048 bits and above in the certificates. Fortunately, we were able to use Elliptic Curve Cryptography (ECC) public keys of lengths up to 256 bits offering security equivalent to that of 128 bit AES or 3072 bit RSA keys [BBB$^+$07] resulting in a packet size of about 800 bytes. While ECC certificates are not as common as RSA or DSA certificates, this was the only option which gave us the possibility to keep the payload size low while using X.509 certificates.

Reliability against packet loss is achieved by retrying the authentication protocol. For connections to TCP services, the server could be configured to reject connections to closed ports by sending a TCP RST such that a failed authentication protocol will immediately result in TCP connection failure at the client which can then immediately retry. Whereas for UDP, the application requires its own protocol for determining the failure or, alternatively a timeout. The optimal value for the timeout would be the sum of round-trip time (RTT) to the firewall, delay for processing the authentication packet and, delay for opening the corresponding port in the firewall.

sKnock Certificates

sKnock uses X.509v3 certificates with the requirement that the certificates' public and private keys should be 256 bits long and generated using Elliptic Curve Cryptography (ECC).

In addition to authenticating the client, the client certificates also carry authorisation information specifying the protocol, port pairs the client is authorised to connect to. This information is encoded in the certificates using X.509v3 extensions under object identifier for Technical University of Munich, 1.3.6.1.4.1.19518 as *Other Name* in the *Subject Alternative Name* (SAN) extension.

sKnock Server

Our implementation of sKnock is developed in Python and works with the Linux *iptables2* firewall. The firewall is dynamically configured to allow traffic from port-knocked connections after the clients are authenticated, while the rest of the traffic is dropped by the firewall, including the sKnock authentication packets. To read the authentication packets we used a raw socket, which in Linux is not subjected to the firewall rules and hence can receive all the traffic reaching the host. As the raw socket receives all the traffic reaching the host, an efficient filtering process is required to filter out valid authentication packets from the rest. This is done by discarding packets which do not meet the following criteria in the order listed:

1. Packet's IPv4 or IPv6 header is valid
2. Packet is UDP and its header is valid
3. Decryption of the packet succeeded
4. Packet has valid sKnock header
    - Byte 32 is 0
    - Timestamp within allowed interval
    - Client IP matches packets source IP
5. Timestamp is within the allowed period
6. Client certificate is valid and not in the revocation list
7. Client certificate authorised for opening requested port
8. Client signature is valid

If the authentication packet passes all of the above checks, the requested port is opened in the firewall for the client sending it.

sKnock Client

sKnock client is an implementation of the sKnock protocol for client side applications. The client implementation is available as a library in Python and C. Applications can use the libraries to perform port-knocking at a sKnock firewall. The libraries contain the following functions:

- *knock_new (timeout, retires, verify, server certificate, client certificate, client certificate password)*: creates a new handle with the given certificates. The fields *retries* and *timeout* specify how many times sKnock should retry port-knocking and how long it should wait before determining a failure; applies to TCP connections. The *verify* flag specifies whether the library should test whether the port has been successfully opened or not; applies to TCP connections. *server certificate* and *client certificate* specify the paths to the server and the client certificates. *client_cert_passwd* is the field containing the password for the client certificate. This function returns a handle which is required by the next function.

- *knock_knock (handle, host, port, protocol)*: perform port-knocking by sending the authentication packet to the *host*. *handle* is the value returned from *knock_new()*. *Port* and *protocol* specify which port should be opened in the firewall after successful port-knocking.

Applications can port-knock a sKnock firewall by using these two functions before they open a connection to a remote service behind the firewall.

In addition to this, the sKnock client implementation contains a command-line helper program to port-knock the server.

4.1.4 Evaluations

Since sKnock is intended as the scalable port-knocking solution, we evaluated its scalability and performance using a variety of tests. As part of this we evaluated the performance of the *iptables2* firewall software, our filtering processes for filtering valid authentication packets, and the latency overhead incurred during connection establishment due to port-knocking.

All evaluations were performed on two workstations running Ubuntu Linux 12.04.5 LTS with 16 GB of memory and a quad-core Intel(R) Core(TM) i5-4590T CPU running at 2 GHz. The underlying cryptographic routines were provided by OpenSSL-1.0.1 and the firewall was iptables2-1.4.12. The machines are connected with a 1 Gbps Ethernet cable when required to form a network.

Firewall

We evaluated the scalability limitations of using iptables2 firewall by measuring the time taken to open 131072 ports (65535 TCP + 65535 UDP) for an IPv4 client. The test adds a rule in the firewall to open a port for the given client and measures the time taken for the firewall to add this rule. This is repeated sequentially for each port until all of them are open for the given client. The evaluation data is shown in Figure 4.2. The results show that the delay in adding new rules is linearly proportional to the number of rules present in the firewall.

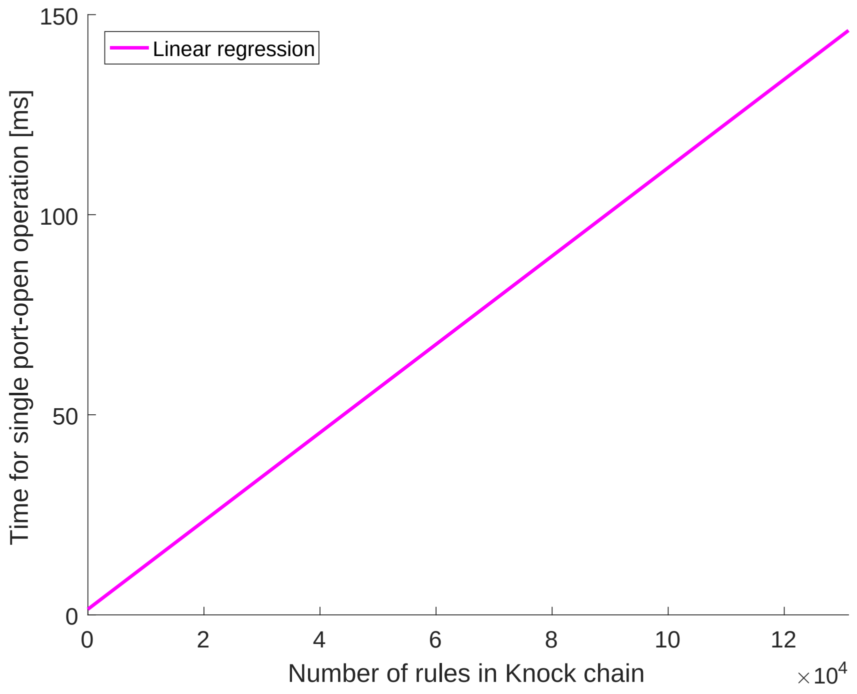


Figure 4.2: Delay in adding new rules to the firewall. Rules are added sequentially to open 131072 (65535 TCP + 65535 UDP) for an IPv4 client.

Packet Processing

Since sKnock uses a raw socket, it has to filter valid sKnock authentication packets from all the traffic reaching the host. For this we defined a filtering process in 4.1.3. In this evaluation we measured the performance limitation of this filtering process with a static test by pre-generating some valid sKnock authentication packets together with some TCP and UDP packets complete with an Ethernet header and random payload. This test does not account for the delay caused by raw sockets as the pre-generated packets are read from a static list instead of raw sockets.

In our test environment, we decided to run two different measurements: one synthetic worst-case scenario, which simulates that all incoming packets are valid sKnock authentication packets and a realistic scenario with 1% of port-knocking traffic while the rest of the packets are irrelevant to sKnock. Additionally the test data for the second scenario contains 5% of packets bigger than the configured minimum authentication packet size. Among these packets the TCP to UDP ratio is set at 5:1 in order to resemble the Internets' traffic patterns as close as possible [ZDJC09].

The test-run performed to evaluate the worst-case processing power of our implementation yielded a result of roughly *5300 pps* (packets per second), which translates to a processing time of less than *0.19 ms* per packet for valid sKnock authentication requests.

Analyzing the performance of our second test case, which is an obviously closer approximation for real-world operation, yields even higher processing capabilities with an average throughput of over *6100 pps* our implementation can achieve computation times of about *0.16 ms* per packet.

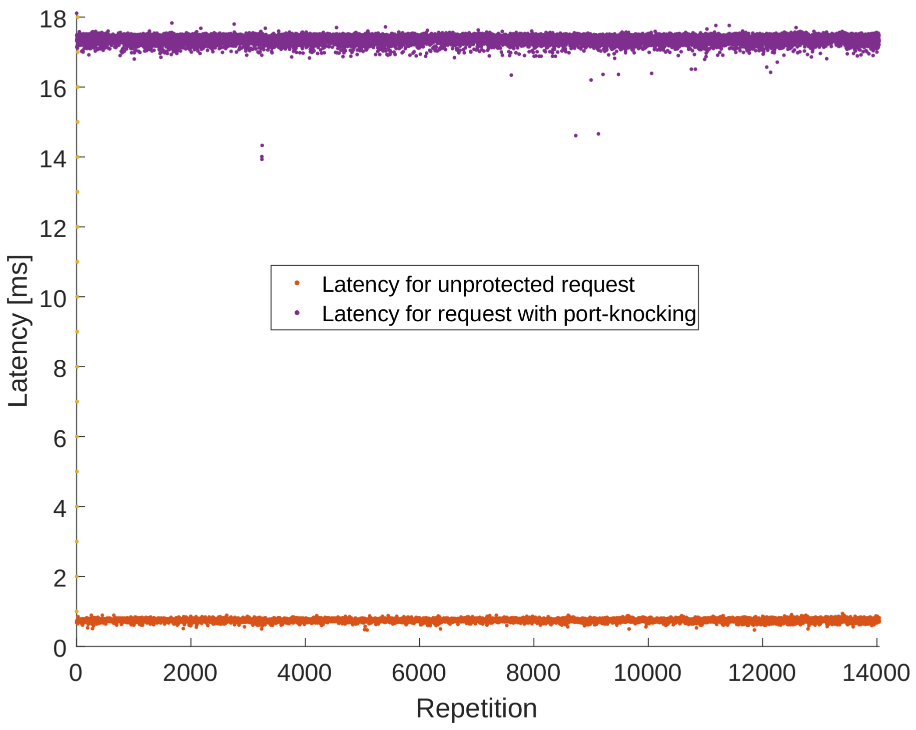


Figure 4.3: Latency overhead caused by sKnock (UDP).

Connection Overhead

In this evaluation we measured the latency incurred while opening connections when using sKnock's port-knocking. We used a simple protocol based on timestamps to measure this latency: the client sends its timestamp to the server as the first packet after port-knocking; the server then responds with its timestamp to the client. The server observes one-way latency while the client observes round-trip latency. The clocks of the client and the server are kept in sync using Precision Time Protocol (PTP)[Wei05].

Since sKnock server requires some processing time to validate a port-knocking request and to add a new rule in the firewall to open the requested port, attempts to connect to a port may fail if the connection packets immediately follow the port-knocking request. Moreover, out-of-order delivery further increases the risk of such failures. To determine the optimal wait period between the authentication packet and the subsequent connection packet, we developed a calibration script. The script tries to open connections with a wait period derived from a start value and retries successful connections with shorter wait periods until a tiny (configurable) fraction of connection open requests fail.

In this evaluation the calibration script yielded an optimal value of 11 ms for the wait period allowing for a failure rate of 0.2% with an accuracy of 99%. To remove noise from the underlying network, we repeated the simple timestamp-based protocol a number of times.

The observed latency for UDP run of the simple protocol without port-knocking protection can be seen in Figure 4.3. The evaluation is also repeated for TCP and the results are summarized in Table 4.1.

Table 4.1: sKnock latency due to connection overhead.

	TCP	UDP
Without port-knocking	1.97 ms	0.74 ms
With port-knocking	18.25 ms	17.37 ms
Port-knocking overhead	16.27 ms	16.63 ms

4.1.5 Limitations

A major limitation of sKnock lies in the amount of overhead caused by the authentication packet. This is far greater than what is required by other implementations. However, if the connection is long-lived, this overhead will be tiny and since only one packet is used, the delay for port-knocking is kept to a minimum. Other limitations of sKnock which we are aware are presented in the following sections.

Incompatibility with NAT

Since the current protocol requires the client to include its IP address into the authentication information, clients using NAT gateways cannot successfully port-knock the firewall. Deployments seeking compatibility with NAT could ignore the client IP check, but they will then be susceptible to A2 attackers.

Vulnerability to Attacks

Any port-knocking scheme employing encryption is further subjected to the *DoS-Knocking* attack [MMETC05] where an attacker carries out a DoS attack by sending many legitimately-looking invalid port-knocking requests to the firewall and keeps the firewall busy while legitimate traffic is left in starvation. We agree that this will be also be the case with sKnock, but due to the usage of raw sockets, only new valid legitimate port-knocking requests are denied service as they are to be processed by sKnock which is kept busy with the DoS requests. Since sKnock's current implementation in Python cannot not occupy all of existing processor cores due to the presence of a global interpreter lock[1], any already opened connections are not starved in this case because the firewall is not processing the port-knocking requests and it has already been configured to allow traffic from previously port-knocked clients.

sKnock is susceptible to replay attacks by {A1, A2, A3, B1} attackers. Such an attacker could wait for the port-knocking to succeed and then take over the connection by masquerading as the client. These attackers can be defended by authenticating the connection data which follows port-knocking as done by Knock[KG14]. This defence is not yet implemented in sKnock.

[1]In the implementation of CPython, the global interpreter lock is a mutex to ensure that the interpreter executes bytecode from a single thread at any time. As a consequence the interpreter cannot allow multiple threads to run simultaneously.

Performance

The current Python implementation of sKnock has limitations in terms of operational efficiency due to Python interpreter being single threaded. While this implicitly gives partial defence towards DoS-Knocking attacks when run on a system with a processor having more than a single core, the throughput could be improved by parallelising request filtering and validation.

Another implementation specific problem with ECC keys is that ECC is relatively new and OpenSSL supports only a limited number of well known curves. Moreover, due to their novel state, there exists no hardware implementations as of this writing to significantly speed up their processing, thus limiting the performance of our implementation.

Trust in NIST Curves

sKnock relies on OpenSSL for providing PKI support. As of this writing, OpenSSL does not yet support any of the curve determined as SafeCurves [BL13] providing 256 bit keys. This led us to use NIST-P 256 curve whose usage may not be secure [BL13] and hence should be replaced when the required support is available in OpenSSL.

4.2 Enhanced Protection for Service Management and Certificates

Public key cryptography and certificates are widely used to protect machine-to-machine communication. WP1 solutions operate in this area and use certificates in a variety of different ways. sKnock uses client certificates to authenticate clients and the services that authenticate themselves towards clients using server certificates. vTTLS improves traditional TLS which is based on server certificates for server authentication. Client authentication is possible in TLS, though less common. The *cryptographic ping* developed for Darshana also relies on certificates for host authentication. Host-to-host communication in Machete can also be protected with certificates.

As SafeCloud aims at a defense in depth approach it is not only desirable to improve the security mechanisms using these certificates, but also consider improvements for the management and use of the certificates and related keys. There is however no general solution for this problem. As the SafeCloud solutions are meant to be used in different cloud environments, the management and AAA (Authentication, Authorization, and Accounting) designs may prefer to attach the SafeCloud software to traditional AAA solutions such as storing certificates in LDAP directories.

4.2.1 Enhanced Protection for Service Management

Secure service provisioning includes setting up and managing services. In a cloud environment these are usually virtual machines, yet it also applies to other cases and physical machines. The management of such machines is done via SSH. Even though the existence of an SSH service is usually to be expected in such an environment, sKnock provides an additional layer of protection for accessing this service. Typical authentication methods for SSH are password or public key based, where client and the service authenticate each other using public key cryptography. While the server is configured to know the client authentication via some mechanism, the overall approach for server authentication follows the *duckling*, also called as TOFU (Trust on First Use) model. So, the first contact to a server is, thus, insecure and key changes or access via other machines will often lead to acceptance of server key changes and thus is vulnerable to potential *man-in-the-middle* attacks.

A recent improvement in this area is the proposal to distribute the server key in addition to internal distribution within the SSH protocol via DNS. The client accessing the server for management will look up the server in the DNS to know its network address. Here, the client in addition to the network address, the client will also receive the server's public key. Combined with DNSSec this provides a secure transmission channel. If DNSSec is not available a VPN connection to trusted DNS servers is recommended. The current GNU/Linux of SSH and DNS already support this mechanism, and the inclusion of Ubuntu software in Microsoft Windows may provide support in a Windows environment as well.

The open issue is to provide a scalable mechanism to export the keys used in the machines in our SafeCloud clouds into the DNS system. For this, we implemented a manager component that manages these keys by interacting with the virtual machines. This manager will then push changes via *nsupdate* protocol into the respective DNS zones corresponding to the virtual machines.

There are two methods of key collection: the light-weight solution is that the manager is configured to know the servers and it will collect the keys via SSH directly. For this to work the network between the manager and the servers needs to be secure and trusted. The alternative option is to have a software and keying material installed on the servers and they will send update messages to the manager when new keys are created.

4.2.2 Certificate Monitoring

The private communications middleware uses X.509 certificates as identities for communication endpoints. The threat that such communication endpoints face when using certificates is a *man-in-the-middle* attack. When an endpoint accepts the rogue certificate of the *man-in-the-middle* attacker, the other endpoint will not notice this. This is because there is no back channel in the infrastructure. The idea of this back channel is to provide information to the endpoints if their correct certificate was seen by their peers. This means that, in case of a *man-in-the-middle* attack, the compromised channel cannot be used directly and another secure channel is needed.

Figure 4.4 shows the scenario in which Certificate Monitoring is meant to operate. There is an application that communicates with other nodes, usually servers belonging to the same application. While communicating the application can also additionally add a small communication with a server in the cloud called MDP to manage the certificate monitoring. The MDP (Manager and Decision Point) is responsible for managing the certificate monitoring of the application and friendly applications in either same cloud or federated clouds. The

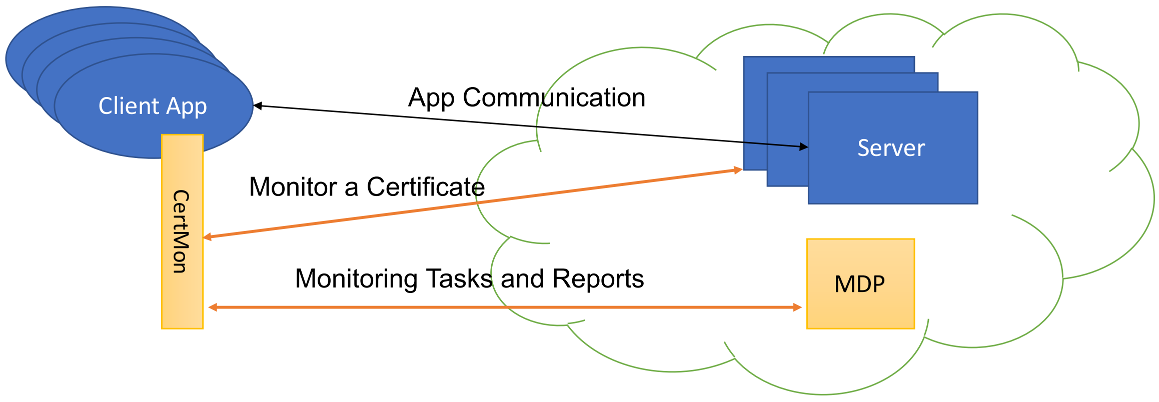


Figure 4.4: Application Scenario for Certificate Monitoring

clients are spread all over the world and might have a different view on the communication to the other nodes, in particular the cloud servers. When they perform monitoring tasks to other nodes, they see a certain certificate. If malicious nodes sit in the middle, they will see a different certificate. With the monitoring and the reporting to the MDP, the malicious certificate may become known to the service via reporting it to the MDP and it can act upon it.

An important related work in this context is the Certificate Transparency (CT) [LLK13] as it tries to make transparent the certificates used in web communication. The idea there is to force the Certification Authorities to publish all the certificate they generate to public ledgers called CT logs. With social pressure and enforcement of this by the web browsers (mainly Google Chrome), presence in the CT logs is about to become a requirement for certificates used in Internet communications. However, this overall approach does not fit to a small or mid-sized cloud application that wants to protect its certificates and be notified on what their clients see (e.g. the app lacks the market power, CT is complicated, lack of direct communication). This leads to a slightly different use case than the general case of a web browser that communicates with unknown arbitrary servers.

The idea of the overall *certificate monitoring* is to find cases where an endpoint might see a wrong certificate because an attacker is interfering with its communication. The certificate monitoring would provide support to check whether other endpoints also see rogue certificates and to find such cases when there is currently no rogue certificate known. The idea is that there is a software component where the application can trigger it to do a request to a service in order to retrieve and report the certificate. To avoid *Denial-of-Service* attacks and related *amplification attacks*, such a scan interaction is limited to one at a time and only triggered by the application. The component will then interact with the manager and report its findings.

Client software will run a monitoring component. When the client software is interacting with the cloud, it will contact the MDP on startup and optionally with low probability contact it in the context of other communication of client and cloud. It opens a TLS communication with the cloud and authenticates the MDP with a pinned certificate. It registers with the MDP and ask for monitoring tasks. The software will store the tasks in a database.

The task processing is done in a randomized way. A random time is selected to wait for some time. Then it is randomly determined if a task is taken or if the another time will be waited. If a task is then selected, the client will open a TLS connection to the requested server. It records the certificate it has seen and stores it in the database. It will send a report to the MDP that contains the seen certificate. Reports will not be immediately sent out, but a processing from time to time checks for new reports and sends them to the MDP. If the sending fails, the report remains labeled unsent in the database and can be sent the next time. The further task processing starts again with a long random waiting period. The motivation for random wait times and the non-immediate interactions is to avoid Denial-of-Service attacks. The main threat is that the MDP has turned malicious and it provides all clients with requests to a victim. The randomization and reduced number of task requested by the client spreads out the requests over time, so that a Distributed-Denial-of-Service attack is avoided.

If client monitoring is used, it seems desirable to consider adding similar functionality to the TLS certificate processing directly. This can be used to add forensics features to TLS and improve handling of revocation.

4.3 Summary

In this chapter we presented the work on protected service provisioning.

We presented sKnock, the port-knocking protection developed as part of the SafeCloud project. We also discussed the enhanced protection of service management and certificates, how they can be used to further protect service provisioning.

The development of sKnock was motivated by the lack of an easy management in the existing port-knocking protections which require the client authentifiers to be kept in sync across all firewalls. sKnock addresses these issues with public-key cryptography using X.509 certificates. sKnock has been demonstrated to our partners Cloud&Heat successfully and it is being tested in their development environments for eventual deployment.

When compared to other port-knocking implementations sKnock has high overhead in terms of payload and processing requirements. Furthermore, stealthy port-knocking with sKnock is not possible. As an advantage, sKnock provides easy deployability to service providers as it can be readily integrated into their PKI infrastructure and as it does not require changes to the client's operating system.

The overhead incurred due to using X.509 certificates could be reduced by implementing a custom format for encoding the certificates into the authentication information. This will however deny us the usage of well-audited PKI libraries, increase development costs and induce security issues. Alternatively, usage of other certificate formats such as OpenSSH certificates could be explored.

During our evaluations we found that the lack of native parallelism in Python limited the performance. In addition to this, there was also some overhead involved in converting data structures between the underlying cryptographic libraries and the firewall interface, which were available as C libraries. Therefore, we believe that on a multi-core system the performance of sKnock could be improved by implementing it in a system programming language such as C or Rust.

Another improvement could be made to add support for UDP data streams. With UDP, the current version requires the client to keep sending authentication packets periodically as long as the application is using the UDP stream to avoid the firewall from timing out the connection and closing the corresponding port. This could be improved by adding application level UDP connection tracking at the server, where the remote service can close the firewall when the connection is no longer required.

Cloud&Heat uses sKnock in their use case CloudBlockStorage which will be presented in the deliverable D5.5 in M36. This use case involves transfer of data between various data centers owned by Cloud&Heat. sKnock is used to provide protection for the service offering the data transfer by allowing the visibility of the service to authorized data centers.

To further protect services, we propose to consider two concepts for enhancing the protection of identities used in the process. We propose that cloud apps should consider certificate monitoring support in order to fight fake certificates and detect attacks against the service.

5 Route Monitoring

The Internet is a network composed by many interconnected networks. Administrative network domains are called *Autonomous Systems (AS)*, and the routing between these autonomous systems is handled by the *Border Gateway Protocol (BGPv4)* [YR06]. Each AS contains one or more *Internet Protocol (IP) prefixes*, whereas each prefix is an identifier for a sub-network. If some AS wants to provide connectivity between its IP prefixes and other ASes, it will announce those prefixes to those ASes. Each AS contains one or more routers configured with BGP, known as BGP speakers. Each speaker contains forwarding tables that provide the information necessary to forward a packet based on the destination and the prefix available in the table. BGP speakers send UPDATE messages to other BGP speakers in order to announce or withdraw routes. Upon receiving these update messages, an AS selects the best route to a certain prefix based on its internal policy.

Although BGP plays an essential role in the Internet, it has considerable limitations in terms of security. One example of its lack of security happened on August 2013 when a company called Hacking Team helped the Italian police regain control over computers that were being monitored by them. Hacking Team worked with an Italian Web host called Aruba announcing to the global routing system 256 IP addresses that it did not own. This caused all the traffic directed to the 256 IP addresses to be redirected to the Hacking Team. This was the first known case of an ISP performing a *route hijacking attack* intentionally [Goo15].

These security problems mainly come from the potential to interfere with route announcements in order to corrupt BGP routing. Attackers can exploit this vulnerability to claim ownership of victim prefixes and announce them to their upstream providers. Providers that do not verify the origin of the announcements may end up injecting these in to the global routing system, which leads network packets to reach incorrect destinations. In some cases, attackers may intercept traffic and forward it to its destination, compromising *confidentiality* without being noticed.

The vulnerability of the BGP protocol has been well-known for over two decades. Several solutions have been proposed, but none is widely adopted and deployed. These solutions mainly fall into filtering and cryptography methods [Lep16, SCK00, WJP03, KPW06], which require changes in routers configurations, router software and a public key infrastructure. Others proposals [LMPW06, KFR06, HMM07, SXW+12] rely on passive monitoring of BGP data, so they are easier to deploy; however, they suffer from high false positive rate, since they access public registries that are frequently outdated. Finally, there are systems that use only data-plane information, by executing active probing, but can easily be bypassed [ZZH+08], or require vantage points [ZJP+07].

We present Darshana (or DaRsHANa, from Detecting Route HijAckiNg – in Sanskrit, Darshana means to see, vision or glimpse) that works by continuously observing network information to detect route hijacking attacks. Our main goal is to detect if Internet traffic is diverted to be eavesdropped in arbitrary places around the world, when the adversary has no access to the path normally taken by the traffic. Therefore, the security property we are most interested in is communication *confidentiality*. Darshana has the advantage of being implemented in the data-plane, above the OSI network layer, therefore it can be implemented in terminals connected to the Internet, instead of being specific to Internet Service Providers (ISP) and other large Internet companies. Darshana uses a set of monitoring techniques like: traceroute, latency measurements and IP traceback mechanisms that can effectively monitor the routes that packets are taking. Ultimately, this allows detecting route hijacking that could be used to eavesdrop on a communication to break confidentiality. We do not intend to substitute the use of best practices to configure BGP, or several prevention mechanisms that have already been proposed like [BFMR10, SCK00, WJP03, KPW06, KFR06, Lep16].

The system applies active probing techniques which enables the detection in near real-time. The order of execution of these techniques is defined in terms of overhead and reliability: techniques with lower overhead and reliability are executed more often; when needed, heavier, more reliable techniques are used. The system does not depend solely on a specific technique to be able to accurately detect attacks.

We performed an experimental evaluation by deploying nodes in PlanetLab and Amazon AWS. We show that nodes can identify when traffic is being hijacked, although this is more difficult if the hijacker is close to the source.

5.1 Background

BGP does not ensure that BGP routers use the AS number they have been allocated, or that the ASes holds the prefixes they originate. Therefore, a router can be configured to advertise a prefix from an address space belonging to another AS in an action known as route hijacking or IP prefix hijacking [BFMR10]. This action can happen in the following forms:

Hijack the entire prefix. The hijacker announces the exact prefix of the victim, meaning that the same prefix has two different origins.

Hijack only a sub-prefix. The offender announces a more specific prefix from an already announced prefix (e.g., the victim announces 200.200.0.0/16, the attacker 200.200.200.0/24). Due to the longest prefix matching rule, ASes that receive these announcements may direct traffic towards the wrong AS.

These forms of attacks can impact routing, leading to:

Blackhole. An AS drops all the packets received. The Pakistan Telecom / YouTube incident originated a blackhole where all the traffic sent to YouTube was redirected to Pakistan Telecom. Since there was no working path back to YouTube, Pakistan Telecom was forced to drop all packets [Bro08].

Interception. The attacker announces a fake route to an AS, that forwards traffic of the victim to the original server. The contents of the intercepted traffic can be analyzed/changed, before sending it to the legitimate destination [BFZ07]. This type of attack requires an untampered working path that will route the traffic back to the legitimate destination.

BGP security procedures today consist mainly on filtering suspicious BGP announcements, e.g., announcements that contain loopback addresses or addresses that are not owned by the AS that announced it. The problem of this approach is that detecting invalid route announcements is more challenging when the offending AS is several hops away.

An accurate routing registry would have prefix ownership, AS-level connectivity and routing policies enabled in each AS, helping ASes to verify the legitimacy of the advertisements that they receive. The drawbacks of this model mainly include, the lack of desire of ISPs to share their proprietary routing policies.

In this work, we focus on interception attacks and propose a solution that does not rely completely on Internet registries.

5.2 Darshana

5.2.1 Mechanisms used in the system

The used mechanisms are presented next. We indicate the short names we use for each between parentheses (e.g., Lat for the first mechanism).

5.2.1.1 Monitoring network latency (Lat)

One of the metrics used in our system is the RTT (round trip time). Each node that is monitoring another (node) keeps information about the total time that each packet takes from source to destination and from destination to source. In a hijacking event the end-to-end latency between a certain source and a destination tends to change significantly. Measuring the RTT has some benefits like low overhead and the fact that time is a factor that is hard for an attacker to evade. On the other hand, an increase in RTT is hard to distinguish from network congestion.

We designed a new version of ping that we denote as *cryptographic ping*. The objective is to avoid having an adversary respond to a ping request earlier, before the request reaches the destination, leading to readings of RTT that are lower than the real value. The new mechanism works as follows. The machine that is monitoring A marks time and sends a nonce to a machine that is being monitored B. B will cipher the nonce with its private key and send it back. A marks the time again and will verify the received signed nonce by applying the public key of B. If the nonce matches, A calculates the round trip time by subtracting the first marked time from the last marked time. Without this ping, the hijacker, since he has hijacked the traffic, could answer to the ping probes sooner, ultimately fooling the system. This way we can guarantee authenticity and uniqueness. This requires the server to run code and share its public key.

5.2.1.2 Estimating hop count (Hop)

We propose adding the hop count, the number of intermediate devices between a source and a destination, as one more criteria to detect a route hijacking attack. According to [ZJP$^+$07] the hop count to a certain destination generally remains unchanged over time. When a prefix is hijacked, the hop count tends to change. In an interception attack, the traffic takes a detour to the AS of the hijacker, then it is forwarded to the legitimate destination. This deviation can change significantly the hop count if the hijacker is far from the source, which is likely due to the size of the Internet. In contrast to the RTT, the hop count is not affected by congestion. However, other less frequent events like link failures and operational route changes may cause it naturally.

5.2.1.3 Calculating path similarity (Path)

The system tracks the path that packets are following. It periodically stores the path obtained using traceroute and translates the IP addresses found to autonomous systems numbers (ASN). This mapping increases accuracy, because we only need one router from an autonomous system to correctly obtain a path that packets are taking. The correlation between the new path measurement and the previous path measurement may provide insights about the occurrence of the attack. In an hijacking event, since the traffic has taken a detour, the paths measured may end up showing significant differences. The level of this difference sets apart legitimate route changes and hijacking situations. On the contrary, legitimate changes are not expected to result in a dramatic route change.

5.2.1.4 Monitoring propagation delay (Prop)

We propose a new technique that isolates the propagation delay from the RTT and uses this metric to declare a route hijacking. This technique is divided into two phases. The second phase is activated only if the system stops obtaining results from the Path mechanism, indicating an attacker is interfering with this mechanism.

Phase one. Consider that the RTT can be decomposed in the following delays: transmission delay (σ_{trans}), propagation delay (σ_{prop}), queuing delay (σ_{queue}) and processing delay (σ_{proc}) as depicted in $RTT = \sigma_{trans} + \sigma_{prop} + \sigma_{queue} + \sigma_{proc}$. The propagation delay is the time that a bit takes in the communication medium from a node to another node. This delay can be calculated as the ratio between the link length and the propagation speed on that medium.

The system uses the IP addresses of the origin and the destination to obtain their approximate geographical coordinates. The link length is calculated by computing the shortest distance between both. For the propagation speed, we make a conservative approximation by considering that all nodes are connected with fiber-optics, which has higher propagation speed than alternative media (copper, air). We use the usual approximation that fiber optics operates at 2/3 the speed of light [opt12]. The minimum possible propagation delay is given by formula 5.1, where *o* represents the origin, *d* is the destination and *c* is the speed of light:

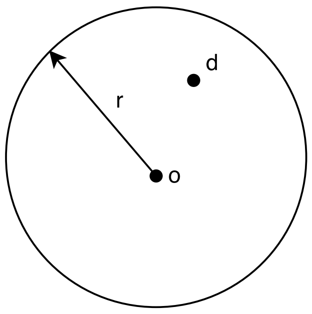


Figure 5.1: The maximum propagation delay is represented as a circle defined by the source *o* with radius *r* where the destination *d* is inside of the circle.

$$\sigma_{prop} = \frac{3}{c} \times \mathit{ShortestDistance}(o, d) \tag{5.1}$$

Besides the propagation delay, the system estimates the sum of the others latencies ($\sigma_{trans,queue,proc}$) by $\sigma_{trans,queue,proc} = \mathit{RTT} - \sigma_{prop}$.

Phase two. When the system obtains an anomalous RTT and stops receiving results from Path, it selects the minimum value of $\sigma_{trans,queue,proc}$ and the maximum value of the RTT is estimated. By $\mathit{max}(\sigma_{prop}) = \mathit{max}(\mathit{RTT}) - \mathit{min}(\sigma_{trans,queue,proc})$, we obtain an upper bound on the value of the propagation delay. This allows drawing a circle around the source with a radius *r* that represents the maximum propagation delay (represented in Figure 5.1). If the distance between *d* and *o* is greater than *r* we detect a route hijacking. This mechanism allows detecting route hijacking even if the Path measurements cease to exist. However, it requires a period of initialization, to estimate the different latencies.

5.2.2 System operation

The operation of Darshana is represented in Figure 5.2 that divides the mechanisms presented in components and presents their relations. Darshana has the following components:

Active Probing. In this component three mechanisms come into play: Lat, Hop, and Prop (first phase). The system constantly takes values for RTT, hop count and the path that packets are taking. The system probes the RTT more often because this is the mechanism with the lowest overhead. Upon detecting an anomaly in the RTT the system passes to more reliable mechanisms, as this anomaly could be caused by temporary congestion in the network. The next mechanism is estimating the hop count, for the reasons explained in 5.2.1.2. This metric is more reliable than Lat so it is used to filter out small legitimate changes. This component also executes the first phase of monitoring propagation delay, calculates this delay with the shortest distance in a straight line between the source and the destination and estimates the other latencies belonging to the RTT.

Path Similarity Detection (Path). Traceroutes with different protocols (ICMP, UDP, TCP) are issued. The system uses different protocols because routers may be configured to block

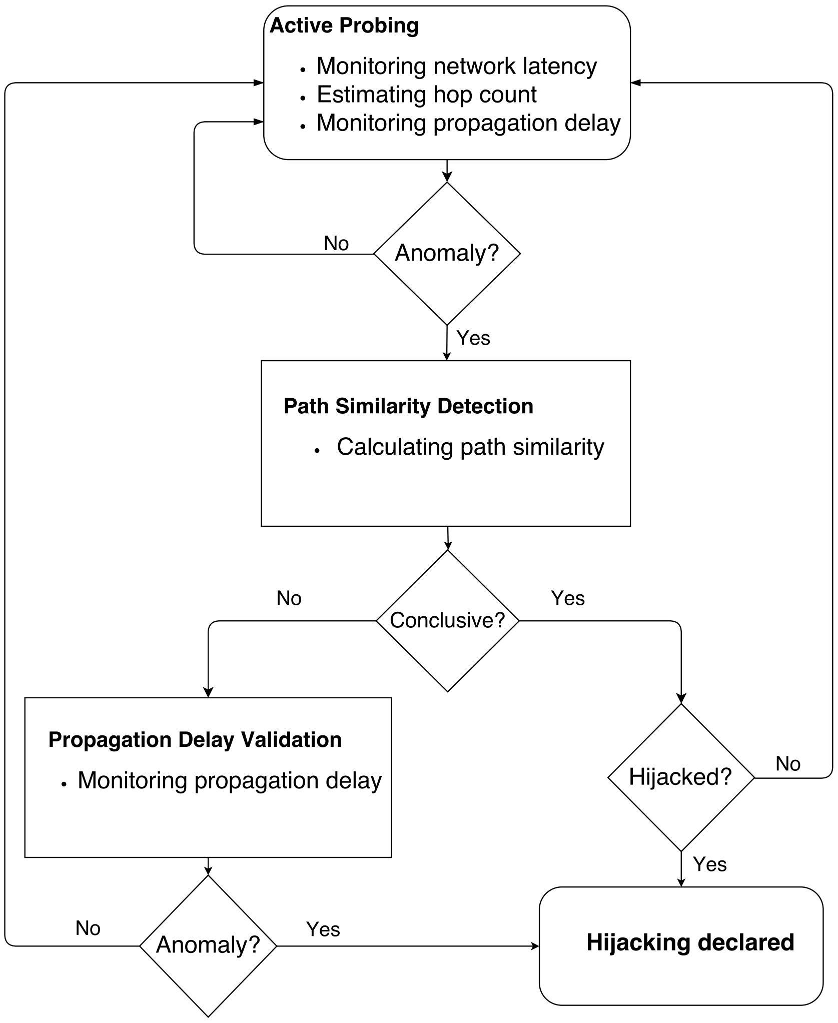

Active Probing
Monitoring network latency
Estimating hop count
Monitoring propagation delay
Anomaly?
No
Yes
Path Similarity Detection
Calculating path similarity
Conclusive?
No
Yes
Propagation Delay Validation
Monitoring propagation delay
Hijacked?
No
Yes
Anomaly?
No
Yes
Hijacking declared

Figure 5.2: Fluxogram of Darshana, with the mechanisms organized in different components.

certain protocols [LHH08]. The path that contains the most nodes is chosen and stored. If enough results were received, then the new path will be compared with the last path obtained by the Active Probing. Disagreement above a certain threshold may indicate the existence of the attack.

Propagation Delay Validation (Prop, second phase). In case no conclusive results are received from path similarity detection, the $max(\sigma_{prop})$ and the $anomalous(\sigma_{prop})$ are calculated. The maximum propagation delay is computed by 5.2 The anomalous propagation delay is calculated with 5.3 This calculated propagation delay is compared with $max(\sigma_{prop})$ by subtracting the values.

$$max(\sigma_{prop}) = max(RTT) - min(\sigma_{trans,queue,proc}) \tag{5.2}$$

$$anomalous(\sigma_{prop}) = anomalous(RTT) - max(\sigma_{trans,queue,proc}) \tag{5.3}$$

Hijacking declared. Upon conclusion of the method chosen, an analysis is made and presented to the sender of the traffic through a notification mechanism. The notification can be handled automatically or presented to a human user.

5.2.3 The system in detail

Active probing. Darshana issues cryptographic pings and Paris traceroute [ACO+06] probes with different periods. Paris traceroute is known to evade anomalies like loops, cycles and diamonds. These anomalies stem from the fact that a load balancer sends probes of traceroute to different interfaces based on the header of the probes. By not varying the fields used by a load balancer, Paris traceroute enables probes to be forwarded in the same interface even if load balancers exist.

Three values are obtained by executing traceroute: hop count, traffic path and propagation delay. For calculating the hop count we use traceroute. We only need to execute a partial traceroute with a TTL that is close to the destination in the majority of times. TTL = 1 is only used when we do not know about the destination.

We characterize the traffic path in terms of a set of autonomous systems, so each node of the result of the traceroute is mapped to the corresponding autonomous system using the CYMRU database [cym].

Finally, the propagation delay is calculated by first, translating the IP of the source and destination to geographical coordinates using MaxMind database [max], then the shortest distance is calculated between them and passed to the propagation delay by using the formula presented in 5.2.1.4.

Each iteration of the cryptographic ping gives a new sample of RTT and by subtracting the RTT with the propagation delay, we estimate the other latencies of the RTT.

Path Similarity Detection. New samples of RTT and hop count are compared with the exponential weighted moving average of past samples, the formula for the average is the following: $sample = (1 - \alpha) \times sample + \alpha \times sample_{new}$. The moving average allows Darshana to adapt to the normal changes in the network. If the quotients between the new samples of both RTT and hop count with the exponential weighted moving average passes certain defined thresholds T_{Lat} and T_{Hop}, Paris traceroutes are issued to the destination in an attempt to reveal the cause of the anomalies. If there are enough elements in the resulting path, then this path is compared to the last path stored. The comparison of these two paths can be computed from *path* and *path′* using the Sorensen-Dice coefficient: $sim = 2|path \cap path'|/(|path| + |path'|)$. This gives the similarity in a number that ranges from [0,1]. 0 means that there is no similarity at all and 1 means that the items of the two paths are the same. If the similarity is below a threshold T_{Path}, then a route hijacking is declared.

Propagation Delay Validation. In case the traceroutes executed in the previous module do not produce any results, Darshana calculates the σ_{prop} with the RTT and $\sigma_{trans,queue,proc}$ that were estimated. More precisely, the system will compute the $max(\sigma_{prop})$, by subtracting the $max(RTT)$, found before the anomaly, and the $min(\sigma_{trans,queue,proc})$. This computed delay will be compared with $anomalous(\sigma_{prop})$ resulted from the subtraction of the $anomalous(RTT)$ with the $max(\sigma_{trans,queue,proc})$. If $\frac{anomalous(\sigma_{prop})}{max(\sigma_{prop})}$ is higher than a defined threshold T_{Prop}, then a route hijacking is declared.

5.3 Evaluation

Simulations of prefix hijackings were conducted to validate our proposed implementation in terms of performance and cost. The objective of the experimental evaluation is to answer two important questions: (1) How effective is Darshana in detecting attacks? (5.3.2) (2) How many times is Darshana forced to execute techniques with higher overhead in normal conditions when there is no attack? (5.3.3)

The experiment was done in PlanetLab Europe [pla] and AWS EC2 [Ama]. PlanetLab offers a geographically diverse set of nodes which provides more choice to build scenarios. However, the restriction of only being able to access nodes from Europe limits the testing in larger scale scenarios. AWS permits access to instances in different continents but does not provide much geographical diversity. With PlanetLab we use nodes from Portugal (POR), Ireland (IRE), France (FRA), Germany (GER) and Poland (POL). From AWS we used instances from N. Virginia (VA), N. California (NA) and South Korea (S. Korea). Figure 5.3 shows a world map with all the nodes used from PlanetLab and AWS marked in black circles and squares, respectively.

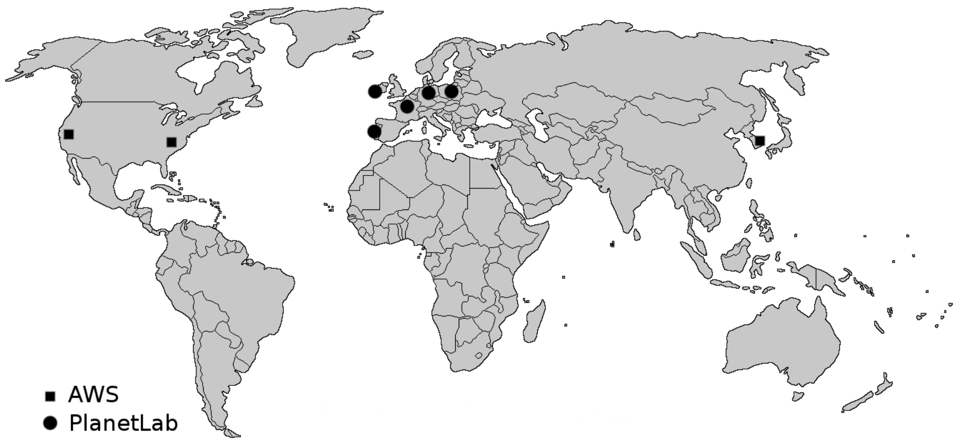


Figure 5.3: Nodes used from PlanetLab and AWS.

5.3.1 Simulating route hijacking attacks

Before we present the tests done, it is important to explain how the simulation of the attacks is made. We simulate only the interception attack, because the blackhole attack ends up being just an interruption of communication, therefore it is easy to detect. In order to simulate the attack, we need three nodes: one node that is the source of the Internet traffic; a node that will serve as the destination; and another node that is trying to hijack traffic by receiving it and then sending it to the legitimate destination.

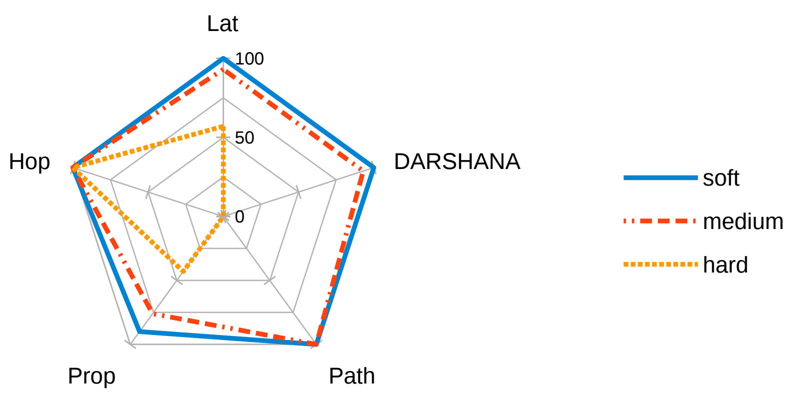


Figure 5.4: The percentage of times that each mechanism detects a simulated route hijacking attack. The scenario involves Portugal as the source, Ireland as the destination and France as the hijacker. The labels soft, medium and hard represent different sets of thresholds for each mechanism.

5.3.2 Performance of the system

In order to evaluate the performance of Darshana, we measured the percentage of times that Darshana detects existing attacks and assess the false positives in different scenarios (i.e., false route hijacks reported). We compared Darshana with the individual mechanism: Lat, Hop, Path, Prop. Each scenario of the experiment was repeated 30 times.

5.3.2.1 Small scale scenarios

We tested small scale scenarios with nodes from PlanetLab. Each scenario is composed by three nodes. Two of them have a source-destination relation and the third one serves as the hijacker. Throughout the scenarios the source and the destination are fixed and the hijacker varies its distance to the source. We selected a node from Portugal as the source, a node from Ireland as the destination and the hijackers are nodes from France and Poland. The distances from source to hijacker were chosen in a way that would enable us to determine the cases when Darshana has more difficulty in detecting the attack. The results are presented in Figures 5.4 and 5.5.

The figures show the percentage of times that each mechanism detects a simulated route hijacking attack under different scenarios. Each figure contains three labels: soft, medium and hard. They specify qualitatively the thresholds that were used for each mechanism. There are four thresholds, T_{Lat}, T_{Hop}, T_{Path} and T_{Prop}, that indicate how much measurements of RTT, hop count, path and propagation delay have to deviate in order to declare a route hijacking. The values of the thresholds used in the experiments were defined based on many experiments done before the evaluation here reported. These values are presented in Table 5.1.

Observing the results, we can conclude that soft thresholds lead Lat to detect all hijacks. However, this also leads to false positives and, in the case of Darshana, prevents the other

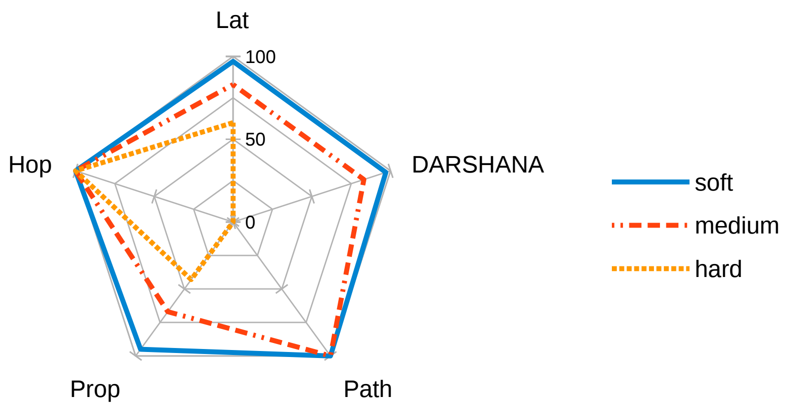


Figure 5.5: The percentage of times that each mechanism detects a simulated route hijacking attack. The scenario involves Portugal as the source, Ireland as the destination and Poland as the hijacker.

mechanisms from actuating and removing such false positives, while keeping a high detection rate. The Hop and the Path mechanism present 0 or 100% values. This is due to the fact that these mechanisms provide constant results through time. Therefore for certain values of T_{Hop} and T_{Path}, these mechanisms will detect or not the simulated attack. In regard to the propagation delay mechanism and observing Figures 5.4, 5.5 and Table 5.1 , the hard label in Figure 5.4 corresponds to T_{Prop} = 4. 6 and the soft label in Figure 5.5 is equal to T_{Prop} = 7.4. In Figure 5.5 with the soft label, the detection of the attack is very close to 100%, but by observing Figure 5.4 we can see that for the hard label, this mechanism can only identify the attack less than 50% of the times. This means that changes in propagation delay are much more observable as the hijacker increases its distance to the source of the traffic.

When the source and the hijacker are close, the packets do not traverse many different autonomous systems and so Darshana is not able to detect the hijack with the hard thresholds.

5.3.2.2 Real scenarios

While our previous analysis provided some insights about the capacity of detection of our system, we wanted to test our detection mechanism using historical prefix hijacking events and confirm that our mechanism behaves better by having the hijacker farther away. We simulated two scenarios. It was not possible to choose nodes from the exact locations in which these scenarios took place so we chose nearby nodes. The first scenario corresponds to the Belarusian Traffic Diversion [bel], where traffic from New York was diverted to Belarusian ISP GlobalOneBel before arriving to the intended destination, Los Angeles. To simulate this, we deployed two nodes (micro-instances) in two different Amazon AWS regions: N. Virginia and N. California. The node from N. Virginia is the source, the node from N. California is the destination. We used a node from PlanetLab in Poland to serve as a hijacker and to represent the Belarusian ISP GlobalOneBel. The second scenario emulates the China 18-Minute Mystery [Cow10], in which, allegedly, traffic between London and

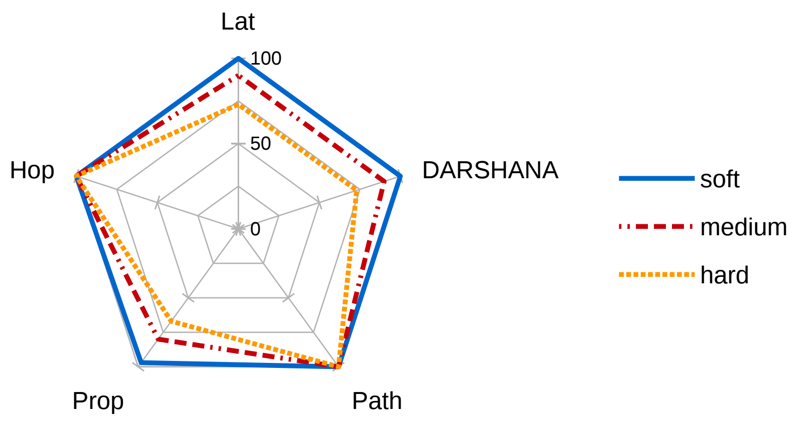


Figure 5.6: The percentage of times that each mechanism detects a simulated route hijacking attack. The scenario involves N. Virginia as the source, N. California as the destination and Germany as the hijacker.

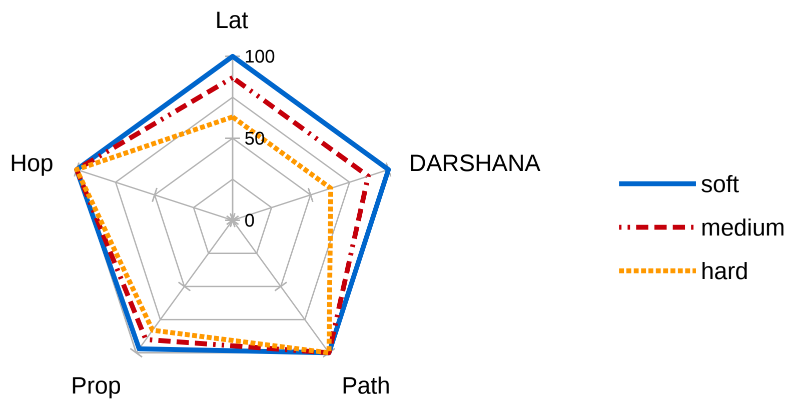


Figure 5.7: The percentage of times that each mechanism detects a simulated route hijacking attack. The scenario involves Ireland as the source, Germany as the destination and South Korea as the hijacker.

Germany took a detour through China. We simulate this by selecting a node from Planet-Lab in Ireland as the source, a node from Germany as the destination and a micro-instance of Amazon AWS in Seoul as the hijacker. The results can be found in Figures 5.6 and 5.7. In these scenarios there is substantially more change, between the samples after attack and the samples prior to the simulated attack, than in the small scenarios experiments. The values for thresholds are shown in Table 5.1.

To better understand why Darshana presents good detection values in relation to the experiments done in 5.3.2.1, we need to have an idea of the paths that packets take from source to destination, before the hijacking and after the hijacking. This information is provided in Table 5.2.

Table 5.2 shows the number of the ASes the traffic traverses, before the attack and after. It is possible to observe that the normal path and the hijacked path from the small scale

Table 5.1: Values of thresholds used for each scenario. S, D and H are the source, destination and hijacker, respectively.

	S:POR I D:IRE I H:FRA			S:POR I D:IRE I H:POL			S:VA I D:CA I H:POL			S:IRE I D:GER I H:S.Korea		
Mechanisms	Soft	Medium	Hard	Soft	Medium	Hard	Soft	Medium	Hard	Soft	Medium	Hard
Lat	1.2	1.3	1.4	2.1	2.2	2.3	3.6	3.65	3.7	13	13.5	14
Hop	1.05	1.1	1.15	1.05	1.1	1.15	2.05	2.1	2.15	2.2	2.25	2.3
Path	0.9	0.8	0.7	0.9	0.8	0.7	0.9	0.8	0.7	0.9	0.8	0.7
Prop	3.6	4.1	4.6	7.4	7.9	8.4	12.5	13	13.5	70	75	80

Table 5.2: Numbers of the ASes that packets traverse.

Hijacker	From - To					
	POR - IRE		IRE - GER		VA - CA	
	Normal	Hijacked	Normal	Hijacked	Normal	Hijacked
FRA	1930,21320,1213	1930,21320,2200,15557,1213	-	-	-	-
POL	1930,21320,1213	1930,21320,8501,8890,1213	-	-	16509	16509,2603,8501,8890,16509
S.Korea	-	-	1213,21320,680	1213,3356,2516, 16509,4766,174,680	-	-

scenarios share more numbers than the paths from real case scenarios.

Furthermore, detecting the attack between two instances of Amazon AWS is easy, because there is not a lot path diversity as we can see from the normal path between N. Virginia and N. California.

5.3.2.3 *False positives*

There is a false positive when a scheme claims to have detected an attack that did not exist. We evaluated the false positives for each individual mechanism of our system during a run of 1h15m. The false positives were calculated by executing each detection mechanism with scenarios without running the attack (i.e., without hijacking). By capturing the amount of alerts given by a mechanism we get the false positive rate $\#alerts/\#samples$, where $\#alerts$ is the number of alerts and $\#samples$ the number of samples taken. We tested for three different scenarios and each scenario contains a source and a destination. For the first scenario, we chose a node from POR as the source and a node from IRE as the destination; in the second, the source is a node from IRE and the destination is a node from GER; finally for the last scenario the source is a node from VA and the destination is a node from CA.

For Path and Hop the number of false positives observed was 0, because there would have to be legitimate route changes to cause them, but these are not frequent and none was observed. For Prop the number of false positives was also 0, as the mechanism always searches for the maximum RTT stored to compute the maximum propagation delay ever observed. Unless a great anomaly in RTT is found, the mechanism will not raise an alarm. For Lat, we received a new sample from 30 to 30 seconds getting a total of 150 samples per scenario. The results are presented in Figure 5.8. The values for the thresholds were chosen with the objective to reveal variation in the false positive rate. As we can see in all sets of columns the false positive rate is bigger for softer thresholds. This makes sense since small thresholds mean that a small variance of RTT is considered an attack. For the soft label the value used was 1.2, for medium the value was 1.3 and for the hard label the value chosen was 1.4.

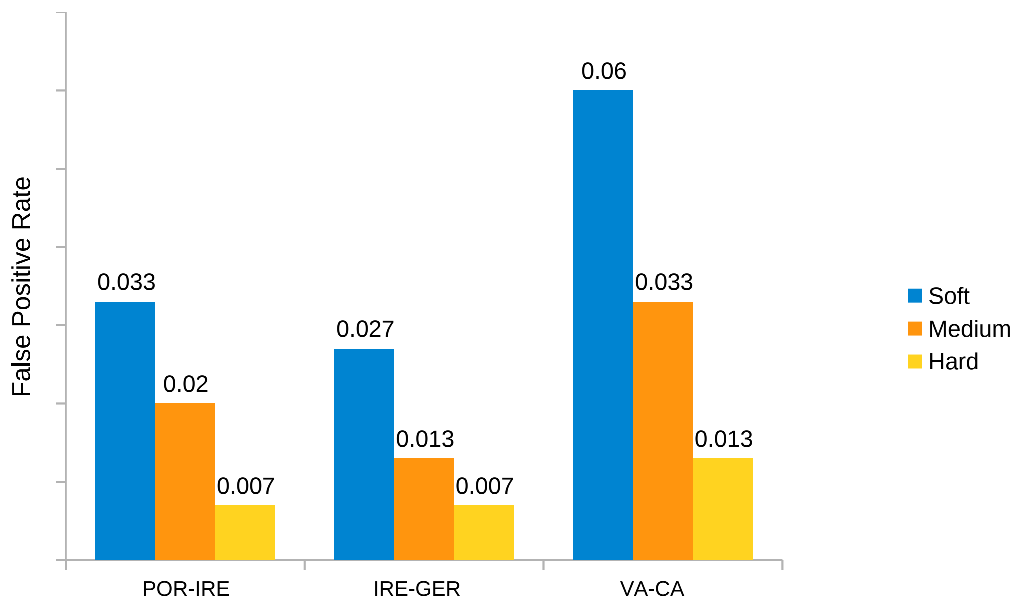


Figure 5.8: False positive rate of RTT in different scenarios. The *Y* axis refers to the false positive rate and the *X* axis represents the different scenarios tested.

The results for Darshana were obtained with the soft thresholds for Lat. However, on the contrary of Lat, the false positive rate for Darshana was 0 in all scenarios, as the other mechanisms (Path, Hop, and Prop) filtered the false positives of Lat, leading to 0 false positives as obtained with each of the 3 individually.

5.3.3 Cost

Darshana keeps probing for RTT with period *k*. Unless anomalies in RTT are verified, leading the system to use techniques with bigger overhead, like Hop and Path. We evaluate the cost as how many times Darshana is forced to execute heavier techniques in normal conditions. The scenarios used were the same as in 5.3.2.3. Setting a probing rate for RTT to 60 seconds, Figure 5.9 illustrates the values for round trip time in different scenarios.

During this period the mean deviations of the samples were low. The scenario with VA and CA, has the biggest mean deviation of approximately 5.02. This implies that the RTT usually remains constant, being difficult to observe anomalies and pass to heavier methods. Considering a value of T_{Lat} = 1.5, the total ping and traceroute messages for this period follow the following formulas, where $\#Msg\_Ping$ and $\#Msg\_Traceroute$ correspond to number of ping and traceroute messages, respectively $\#Msg\_Ping = T \times k$ and $\#Msg\_Traceroute = \#Msg\_Ping/n$.

Where T is the total time of the experiment, k is the ping period and n is the traceroute period. For this experiment $\#Msg\_Ping = 100 \times 1 = 100$ ping messages and $\#Msg\_Traceroute = 100/5 = 20$ traceroute messages. All of this demonstrates that even for low values of T_{Lat}, the total number of $\#Msg\_Ping$ and $\#Msg\_Traceroute$ end up only being dependent on k and n.

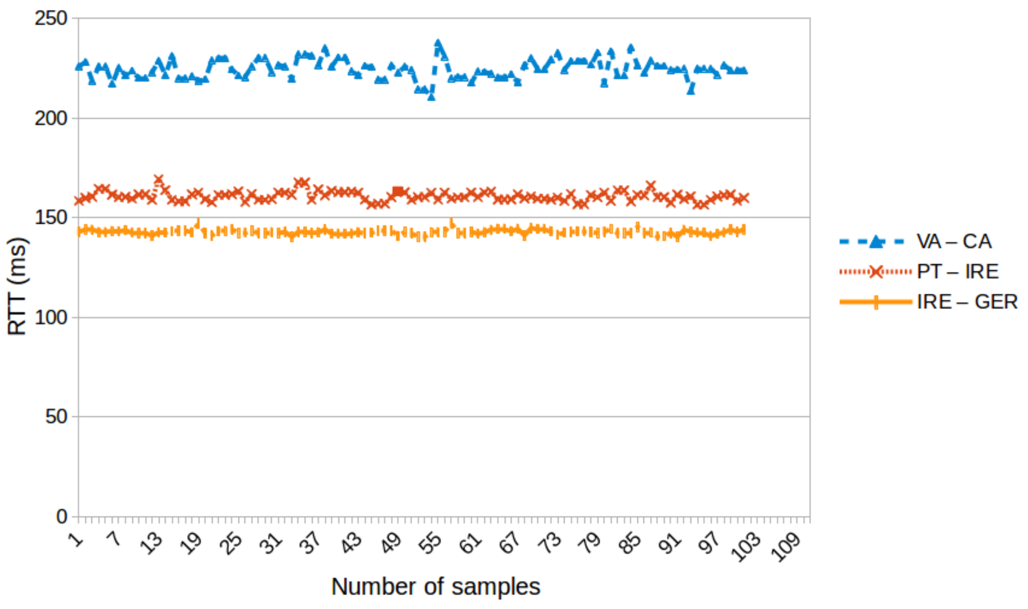


Figure 5.9: RTT values in different scenarios. The *Y* axis refers to the RTT values in milliseconds and the *X* axis represents the number of samples.

5.4 Related Work

Many solutions have been proposed for the IP prefix hijacking problem. Some of them are crypto-based such as [Lep16, SCK00, WJP03, KPW06]. These solutions require deep changes in routers and network protocols. BGP routers need to sign and verify announcements which leads to a non negligible overhead.

Other solutions like [LMPW06, KFR06, HMM07, SXW+12] are more deployable because they do not require changes in routers, they only need access to public registries, like Route Views and European IP Networks (RIPE) to conduct passive monitoring and look out for Multiple Origin Autonomous Systems (MOAS) [ZPW+01]. An IP prefix should only be generated by a single AS, so this conflict may indicate a prefix hijacking. The problem associated to these solutions is that many times the public registries may be outdated and inaccurate, leading to an increase of the number of false positives.

Finally, there are solutions that rely only on the data plane like ours. They are not constrained by the availability of BGP information and are more accurate. [ZJP+07] uses a set of monitors to detect prefix hijacking in real time. These vantage points monitor a prefix from topologically diverse areas. Each monitor keeps track of the hop count and the path to a target prefix and if past measurements disagree with new ones then a route hijacking is declared, the need for vantage points end up limiting the system. On the other hand, [ZZH+08] detects IP prefix hijacking by observing unreachability events. It is owner-centric, in a point that the mechanism keeps sending probes to transit ASes. If enough ASes stop responding, the system declares the attack. If the attacker forwards the responses of the probes back to the sender, the attack is not detected.

Here we make use of the propagation delay as another criteria to detect route hijacking. This delay has been used in [LLV+15], which proposed a system that presents undeniable proof about traffic traversing a certain forbidden zone defined by the sender. To know if a certain relay node is not in the forbidden region, the minimum possible RTT from the source to any node in the forbidden zone was calculated, with the propagation delay. In case the RTT from the source to the relay node is less than the RTT calculated earlier, then the relay node is not in the forbidden region.

The design of the lightweight and end-host-based probing techniques was inspired by Hubble [JKBM+08], where low overhead probing techniques are used first and heavier, but more reliable techniques, are only used when there is such a need.

5.5 Summary

This chapter presented Darshana, a route hijacking detection system. By only applying active probing methods, we ensure accuracy and deployability. Different techniques turn the system redundant enough to not be avoided by attackers. The design of the detection system minimizes the overhead, by using techniques with low overhead more often. Techniques with greater reliability and overhead are only executed when necessary. Our system is the first to use the propagation delay in this context, providing one more metric for the purpose of detection. We evaluated the system with small scale and real scenarios.

6 Multi-Path Communication

Sending information over the Internet has the disadvantage of making it vulnerable to eavesdropping by unauthorized third parties. This problem is especially important for organizations that handle critical data, such as governments, military, or healthcare.

Communication protocols based on cryptographic mechanisms such as HTTPS and IPsec are the common solution to this problem. However, recent events show that it may be possible to break these protocols under certain conditions, and suggest that powerful adversaries may be able to do it if they access the encrypted data. For example, Adrian *et al.* presented a flaw in the Diffie-Hellman key exchange that allows downgrading the security of a TLS connection for a specified 512-bit group [ABD$^+$15]. They claim that a nation-state may have the computational power to attack 1024-bit groups, which would allow decryption of many TLS channels over the Internet that implement this method.

We present *Multi-pAth Communication for sEcuriTy* (Machete or MAChETe), a means to mitigate the impact of such vulnerabilities. This system consists on using *Multi-Path TCP* (MPTCP) [FRH$^+$11, BPB11, PB$^+$] and *overlay networks* [ABKM01, AD03] to split communication flows on different physical paths, possibly over a multihomed subnetwork [AMSS08, AMS$^+$03], as a defense-in-depth mechanism.

The rationale is that more effort is required to eavesdrop data split over several flows, in comparison to a single end-to-end flow. The problem addressed in this chapter is, therefore, achieving communication *confidentiality* assuming the confidence on cryptography mechanisms not containing vulnerabilities is not enough due to the criticality of data being sent.

Machete has to handle two challenges. The first consists in sending packets over *different paths* when Internet's routing imposes a single path between a pair of source and destination network addresses. Overlay routing enables doing *application-layer routing*, allowing packets to deviate from the routing imposed at network level, by the Internet's routers and routing protocols. Overlay networks, possibly in combination with multihoming, are used to create path diversity, allowing flows to be split over physically disjoint paths. Using a topology-aware decision algorithm, several *overlay nodes* are chosen, according to their location. Each node will create a single-hop overlay path to the destination, generating an overlay network.

The second challenge is to split the stream of data sent over a TCP connection. MPTCP is a recent extension of the TCP protocol that has the ability to distribute and send data among the different network interfaces of a device, e.g., the IEEE 802. 3 ("wired", "Ethernet") and the 802.11 ("wireless", "WiFi") interfaces of a personal computer. However, MPTCP neither ensures the use of different physical paths, not their diversity, as it was created mostly with performance in mind. The paths used by a MPTCP connection are imposed by the network interfaces of the source and destination hosts.

The combination of MPTCP with application-layer routing is itself a third challenge. Our objective is that Machete works at the *application layer*, without modifications to lower layers, but it has to route packets sent at transport layer under the control of MPTCP. MPTCP is a transport-layer protocol, so applications provide it source and destination IP addresses and ports. However, the overlay nodes have their own IP addresses and ports, unrelated to the previous ones. Therefore Machete has to play with the destination IP addresses and ports for communication to be possible.

6.1 Background and Related Work

The current Machete concept covers the use case where only few relays are available. This corresponds to the current situation. If the technologies of SafeCloud experience a breakthrough in usage, relays in many networks all around the world could be possible. We studied potential benefits of multi-hop multi-path concepts, in particular different options for diverse path selection algorithms, most prominently Suurballe's algorithm[Suu74]. This can be used for a more controlled multipath routing and to support geo-diverse routing[CGL+14]. On the basis of a large-scale traceroute scan, we showed that topology-aware algorithms based on Suurballe's algorithm can improve multiple metrics for resilience like *Effective Node Diversity* or *Failure Probability* [HNC18].

The most common solutions with respect to network resilience operate in the context of improving resilience of an ISP (Internet Service Provider) network. This means that all routers can support the protocol and, thus, there is no limited number of relays. geoDivRP [CGL+14] is optimized to route around large-scale geographic outages. This is achieved by requiring enough geographic distance between different paths in the multi-path. A different approach is multi-topology routing where the routing protocol is run k times with different link costs for each run [MM05].

The following sections cover background and related work on the mechanisms used in Machete: Multi-Path TCP, overlay routing, and multihoming.

6.1.1 Multi-Path TCP

Multi-Path TCP (MPTCP) is an extension of the TCP protocol that enables endpoints to use several IP addresses and interfaces simultaneously when communicating [FRH+11, BPB11]. The protocol discovers which interfaces are available for use, establishes a connection, and splits the traffic among them. It presents the same programming interface as TCP, however the data is spread across several flows. The option field in the regular TCP protocol is filled with MPTCP data structures in order to inform the other end-point about the capability of implementing this protocol and to add flows to the communication. MPTCP has two important components on its configuration: path manager and packet scheduler.

The *path manager* is the module that handles how the flows are created in an MPTCP connection. The implementation of the protocol in Linux currently implements four schemes [PB+]: `default` does not create any new flows, but accepts those incoming; `fullmesh` creates a full-mesh of flows with all available interfaces/addresses in the device; `ndiffports` takes only one pair of IP addresses and modifies the source port to create the number of flows set by the user; `binder` uses the loose source routing algorithm of the Binder system

[BFM13].

The *scheduler* handles the distribution of the TCP packets (segments) over the flows, in close collaboration with TCP's congestion control mechanism [WR11]. MPTCP does not use a single congestion window as TCP, but one per flow. Similarly to TCP, the congestion control mechanism manages the size of each congestion window based on the round-trip time (RTT) of the flow and other factors (timeouts, reception of acknowledgments). The implementation of MPTCP for Linux by default fills the flow congestion window before starting to schedule packets on the next flow. Although in terms of performance it is important to take advantage of the throughput of the channels, in terms of splitting data for confidentiality it may be a disadvantage. In a communication composed by two flows where one has twice the throughput of the other, that flow will tend to send twice the amount of data of the other. If that flow happens to be eavesdropped, a higher amount of data is susceptible to being spied upon. Linux's MPTCP implementation provides three scheduling modes:

- `default`, the one we just presented and the one with best performance; only uses another flow if the window of the flow in use does not allow sending data that is pending; starts sending using the flows with lower RTT;
- `fast round-robin` which uses sequentially all flows but fills the congestion window of a flow before starting with the next;
- `strict round-robin`, does real round-robin by sending the same amount of data through all the flows in sequence; waits for a flow to have free space to send a packet before scheduling the next one.

MPTCP is very similar to TCP in terms of security. Specifically, the RFC says that *"The basic security goal of Multi-Path TCP (...) can be stated as: provide a solution that is no worse than standard TCP"* [FRH+11]. There are a few works concerned with the security of MPTCP [DBVV11, Bon14].

6.1.2 Path Diversity

Path diversity can be achieved in multiple ways in a multi-path communication. *Overlay Routing* and *Multihoming* are among some of the options.

Overlay routing allows the creation of a virtual network (an overlay network) on top of an already existing network infrastructure, like the Internet, without modifying it. The nodes of the network are hosts, i.e., machines that implement the network stack up to the OSI application layer. An overlay link may connect two nodes either directly or indirectly, through other nodes. These nodes route (forward) the packets at the application layer to the next or final node of the link. This adds a level of indirection in relation to the underlying OSI network layer topology. At the network layer the packets travel through the routes imposed by the Internet routing protocols, namely the Border Gateway Protocol v4 (BGPv4) [RL95]. At application layer the traffic may be deviated from these routes by sending it through overlay nodes in other locations. The original motivation for overlay routing is resilience [ABKM01, AD03]. In case a network layer route is congested or faulty, routing done at the application layer may allow passing the traffic through other routes.

Another method of achieving path diversity is to use *multihoming*. This approach consists of having a customer network linked to two or more ISPs, instead of a single one. Resilience and performance are the main advantages of this approach [HR08]. Different providers offer different performance levels to different parts of the network, so choosing the "right" provider will result in a performance increase [AMSS08, AMS$^+$03]. Akella *et al.* [AMS$^+$03] evaluate the use of multihoming, using Hand-shake Round Trip Time (HRTT) as a measurement unit for data centers and enterprises. In [AMS$^+$03], they conclude that simply by using two providers the performance is increased by at least 25% and that the improvements are very small beyond four providers. The same authors in 2008 [AMSS08] performed a similar but deeper k-homing study, measuring the performance based on the RTT and throughput of small (10 KB) and medium sized (1 MB) file transfers. In general, the same conclusion is reached in all multihoming studies: the use of two or three providers increases the performance in 15-25%.

6.2 Machete

Machete is an application-layer mechanism for improving communication confidentiality by splitting packets in different paths. It uses MPTCP over an overlay network to create multi-path communication. By setting up a network composed by several nodes it is possible to implement an overlay network, which consists of several links between the source and destination of a communication. MPTCP will use these overlay links to split the data to be transferred. Machete uses single-hop overlay routing as there seems to be no gain in using more hops. Path diversity is sought by exploiting diversity between Autonomous Systems (ASs).

The architecture of Machete is represented in Figure 6.1 and has three main components: multi-path devices which are the devices that communicate using Machete that also can play the role of server (wait for connections) or client (start connections), similarly to TCP; Overlay nodes which are the nodes of the overlay network that forward messages on behalf of multi-path devices and create alternative communication paths; multi-path manager which is the component in charge of keeping track of the nodes that compose the overlay network.

6.2.1 Threat Model

Machete is concerned about attacks against the confidentiality of data exchanged, so it considers passive attackers that eavesdrop on communication at certain physical locations. We assume that the attackers can eavesdrop on all packets at those locations, so confidentiality has to be achieved by reducing the locations where all traffic passes.

We assume that the attackers do not modify the packets exchanged compromising the communication integrity, but protection against such attacks might be implemented on top of Machete. For example, message integrity codes or keyed hashing might be used [KBC97].

We assume that the devices and nodes of the system are trustworthy, i.e., that they follow the protocol. This assumption has to be assured using proper security mechanisms, such

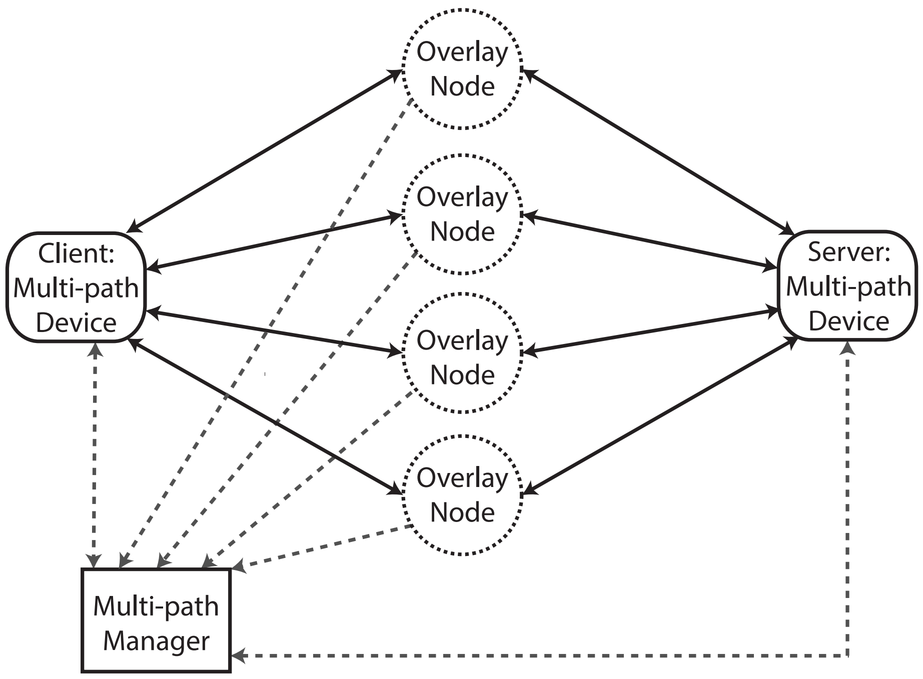


Figure 6.1: The architecture of Machete. The solid lines represent data communication flows and the dashed lines control communication (e.g., messages to perform node registration in the overlay network).

as hardening, sandboxing, and access control. Machete is, however, prepared to recover from node crashes.

The multi-path manager might be a single-point of failure of the architecture, so it is replicated. We assume that a subset of the replicas can be compromised by an attacker or crash and we use a specific scheme to make overall multi-path manager tolerate these issues.

6.2.2 Multi-Path Manager

The multi-path manager is the component that contains information about every entity in the network. Its function is to register every node and device addresses and to provide that information to devices that aim to communicate.

The multi-path manager was not developed from scratch but instead is a *tuple space* that implements Linda's generative coordination model [Gel85]. A tuple space is a repository of data items called *tuples* and provides mainly three operations: insert tuple (`out`), read tuple (`rd`), and remove tuple (`in`).

Machete uses a specific tuple space called DepSpace [BACF08, dep]. DepSpace is replicated in order to tolerate faults in some of the replicas. Specifically, it continues to operate correctly despite the failure of up to f out of $3f + 1$ replicas (typically 1 out of 4). DepSpace is Byzantine fault-tolerant, so it provides its service correctly even if f replicas are compromised or fail arbitrarily. Whenever server multi-path devices and overlay nodes start to run, they register with the multi-path manager by inserting on the tuple space.

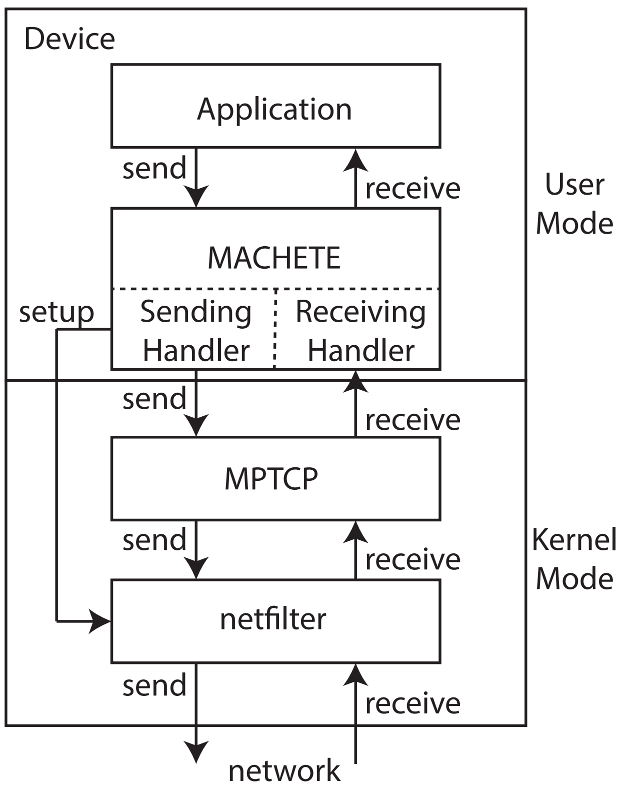


Figure 6.2: Machete multi-path device architecture. Machete is a user level process running in a computer, which connects the application to the communication stack in the kernel (MPTCP) and netfilter (using `iptables`). Machete only establishes new rules when it creates a new connection (essentially an MPTCP connection).

6.2.3 Multi-Path Device

We use the term multi-path device to designate a computer that uses Machete to communicate. The architecture of such a device is represented in Figure 6.2. This component dynamically establishes paths and splits the packets among them.

After a device registers itself on the multi-path manager, the process of transferring a stream of data (e.g., sending a file) is composed of three steps: *path setup*, *data transfer* and *path tear down*. Figure 6.3 represents this process. Next we describe each of these steps, dividing the first in two substeps.

MPTCP requires devices to have several network addresses to create more than one flow. If the device has several physical interfaces, possibly connected to more than one provider – *multihoming* –, each one has an IP address. If that is not the case or that number of addresses is not enough, more than one address can be assigned to each interface, e.g., using Linux's virtual network interfaces [KKJ01]. Having two addresses (in total) on each device is enough to establish a network composed of four paths, which in general is enough to achieve the objective.

6.2.3.1 Path setup -- choosing overlay nodes

The process starts by querying the multi-path manager about the available overlay nodes. Although the manager replies with all nodes available in the network, the number of nodes to be used by a certain connection, N_n, is a configuration parameter.

The overlay nodes are chosen taking into consideration the path *diversity* they provide. If there are several paths with the same diversity, the path with best *performance* (e.g., lowest RTT) is chosen. In the current version of Machete, the metric of diversity among two paths used is the number common ASs on both paths (higher number means worse diversity). For a path, the ASs are obtained using layer-4 traceroute [lft], which provides precisely the ASs of the nodes along a path. The metric of performance is the RTT, measured using the `tokyo-ping` tool, which avoids some anomalies in `ping` [PCVB13]. When available, multi-homing tends to improve diversity as the first ASs along the path will already be different, whereas with single-homming the opposite is true.

In Machete the path manager is set to `fullmesh`, to allow defining the number of flows in a way that makes MPTCP use the number of overlay nodes defined (N_n). This manager will create a network mesh composed by all the available interfaces/addresses in both the source and destination.

To balance the data among all nodes and obtain the expected confidentiality, the best packet scheduler is `strict round-robin`. This scheduler is configured with the number of packets sent in each flow before passing the turn to the next flow. To reduce the information sent in each flow (thus in each path), this parameter is set to 1. The `fast round-robin` scheduler can also be used if the communication is encrypted and the amount of bytes sent is high enough to ensure that not all communication passes in the same node, as it is not possible to decrypt data if it is not complete. This aspect of the amount of bytes sent being enough or not to make the communication pass in more than one node is analyzed in 6.3.3.2.

6.2.3.2 Path setup -- managing addresses and ports

As already pointed out, the combination of MPTCP with application-layer routing is challenging. Machete works at application layer but it has to route packets sent by, and under the control of, a lower layer protocol: MPTCP, at transport layer.

Similarly to what happens with TCP, in MPTCP all packets sent over a connection take two pairs of IP addresses and port numbers, one for the source device, another for the destination (the difference in relation to TCP is that source and destination may have more than one address/port pair). However, in Machete the destination address and port may have to be different: (1) if the packet is leaving the sender device, the destination address/port should be those of the overlay node for the packet's flow; (2) if the packet is leaving an overlay node, the destination address/port should be those of the destination device and the source address/port should be those of the overlay node. (3) if the packet is returning to the overlay node, the destination address/port should be those of the source device and the source address/port should be those of the overlay node.

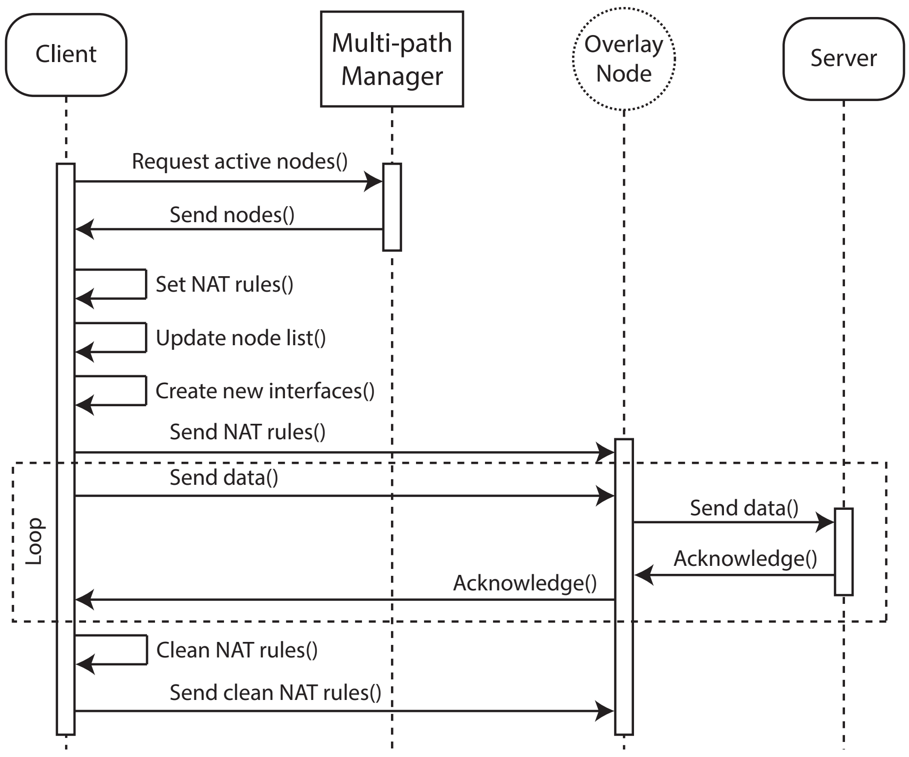


Figure 6.3: Machete communication setup and data transfer example with a single overlay node.

The application requires Machete, thus also MPTCP, to send packets to the destination address and port. When a device does the setup of a path, it has to force these alternative addresses and ports to be used. To do it Machete leverages Linux's *netfilter* framework and the `iptables` command [Rus02]. This framework allows doing network address translation (NAT), packet filtering, and other forms of packet handling. Machete uses it for NAT.

When a path is setup, Machete uses the `iptables` command to tell netfilter to change the destination IP address and port by those of an overlay node, depending on the flow (case (1) above; arrow *setup* in Figure 6.2). MPTCP inserts the destination IP address/port in the packets, but netfilter exchanges them before they are transmitted into the network. The `iptables` command inserts NAT rules for that purpose in the `output` chain, which is the set of rules applied to traffic being sent by a computer. For each link, a NAT rule is set[1].

Once this is done, the device informs each node about the rules they have to establish. In the overlay node it is necessary to route the traffic in both directions: when forwarding to the server (case (2) above) and when returning to the client (case (3) above). As soon as all nodes confirm that the rules are set, the data transfer may begin.

Figure 6.3 shows a time diagram that represents this process with a single overlay node.

[1]The format of the `iptables` rule is: `iptables -t nat -A -p tcp -s <source address> -d <destination address> -j DNAT --to-destination <new destination address>`

6.2.3.3 Data transfer

Machete uses MPTCP to establish a connection to the destination device. The client application will create a socket and provides it one of the server's address/port pairs; the MPTCP protocol will handle the passive creation of the flows. Despite the fact that netfilter modifies the destination addresses to deviate the connection's packets through the overlay nodes, the connection and each of its flows end up established similarly to what would normally happen with MPTCP.

This connection has two data streams, one in each direction, so that the client and server can use to send data to the other. This is represented in Figure 6.2 through the send and receive arrows. Notice again that the scheduler should be set to `round-robin`, otherwise MPTCP will fill each flow until its congestion window is full instead of sending packets using all flows, which is not desirable from the confidentiality point of view.

Each packet will suffer changes on its source and destination address twice: first in the source device, second in the overlay node. The same will happen to the acknowledgement packets.

6.2.3.4 Path tear down

To terminate a connection, the client device notifies the nodes that compose the overlay paths to remove the rules. The overlay device is listening on a specific port for receiving this indication, so that the packets destined to the node itself are never re-routed. Again, this device waits for all nodes to reply before removing its own rules. After all the steps are done, the communication can be declared as finished. If the client fails to inform the nodes about the rules removal, the rules can stay established, since it is specific for a pair of source and destination addresses and, therefore, does not modify other connection's correct behaviour or the possibility for the same source to create an identical connection.

6.2.4 Overlay Node

The overlay node is the component that plays the role of application-layer router, i.e., which forwards the packets received from the client device to the server device and vice-versa.

Overlay nodes receive from clients NAT rules and add/remove them from netfilter. These rules are set, again, with the `iptables` tool, this time using the `prerouting` and `post-routing` chains. The first chain leverages the changes on the traffic immediately after it was received by an interface and the second leverages the changes right before it leaves. For each overlay network, four rules are established, two to change the source and destination when forwarding to the destination and two when forwarding to the source, as mentioned above in cases (2) and (3).

Pair of paths	Common ASs	Common ASs (except first 7)
Singapore, Tokyo	13	6
Frankfurt, Seoul	12	5
Frankfurt, Tokyo	12	5
California, Seoul	12	5
California, Tokyo	12	5
Oregon, Tokyo	12	5
Seoul, Tokyo	12	5
Seoul, Virginia	12	5

Table 6.1: Least diverse pair of paths in terms of number of common ASs with the single-home configuration. The paths are designated by the location of the overlay node.

6.3 Experimental Evaluation

We placed hosts in the Amazon AWS EC2 service [Ama] in nine different regions (Ireland, Frankfurt, North Virginia, California, Oregon, Tokyo, Seoul, Singapore and Sydney) and one in Portugal. We used up to 8 overlay nodes, one in each of the AWS regions, except for Ireland that contains the server. Moreover we placed the client in the Portugal node. Therefore, between the client and server there are 8 single-hop overlay paths: one per overlay node.

Recall that the objective is to provide confidentiality by splitting communication over physically diverse paths with an acceptable performance. Therefore, the evaluation provides an assessment of the diversity in our scenario, presents a performance benchmark of the system, and analyses the confidentiality achieved.

6.3.1 Diversity

As stated before, confidentiality is only achieved if the paths are topologically disjoint, as attackers eavesdrop on traffic at certain locations (6.2.1). The approach used to verify the topology of the paths is to trace each route's chain of ASs from the source device to each node and from that same node to the destination. For that purpose we use layer-4 traceroute (i.e., the `lft` tool).

Table 6.1 shows the number of common ASs in the pairs of paths with highest value, between the 8 single-hop overlay paths, where each is designate by the location of the overlay node. There are at least 7 ASs in common in all paths leaving the client (Portugal). The reason for this lack of diversity is the fact that we did not use multihoming. Moreover, several ASs belong to Amazon, as also expected.

We did an additional experiment to confirm that multihoming is beneficial in terms of diversity. We connected a second interface of the client device to a public provider of 4G service through a smartphone, then we used `lft` to obtain the ASs traversed by the paths. As shown in Table 6.2, using multi-homing provides an evident diversity, where the common nodes are again part of AWS's network. Notice that we used this multihoming con-

Pair of paths	**Common ASs single-homed**	**Common ASs dual-homed**
Oregon, Sydney	10	3
Oregon, Tokyo	12	3
Oregon, Seoul	11	2
Sydney, Tokyo	10	2
Frankfurt, California	9	1
Frankfurt, Oregon	10	1
Frankfurt, Seoul	12	1
Frankfurt, Singapore	11	1

Table 6.2: Least diverse pair of paths in terms of number of common ASs with the dual-home configuration, in comparison to the single-home configuration. The paths are again designated by the location of the overlay node. In the dual-home configuration the left path uses the original connection and the right the 4G connection.

figuration only for this test; the single-home configuration was used in all the experiments presented in the following sections.

In the first configuration still some level of diversity is achieved considering most nodes are in AWS, except for the first 7 common nodes. However, multihoming reveals to be a key component for Machete to achieve path diversity.

6.3.2 Performance

The performance evaluation considers three different aspects: the impact of adding paths on the delay of transferring files, the performance with diverse paths when transferring files, and the performance of path set up and tear down.

Figure 6.4 provides some insight on the network by showing the latencies between the hosts in the different locations, obtained with the `tokyo-ping` tool.

Machete forced MPTCP to use all the paths defined for every experiment by changing the number of IP addresses at the client (i.e., Portugal): 2 addresses for 2 paths, 3 addresses for 3 paths, etc. The client had a single network interface; the server had a single interface and a single IP address. All measurements were repeated 30 times.

6.3.2.1 Impact of adding paths

The evaluation consisted in observing the performance when paths (equivalently, nodes) were added one by one based on latency to the cluster: first in Frankfurt, next in N. Virginia, California, Oregon, Tokyo, Seoul, Singaporeand finally Sydney (that has a the highest latency, as observed in Figure 6.4). The size of the files varied from 1 Byte to 1 GByte. In this experiment we used the `fast round-robin` scheduler to improve performance (and the `fullmesh` path manager which is fixed for Machete).

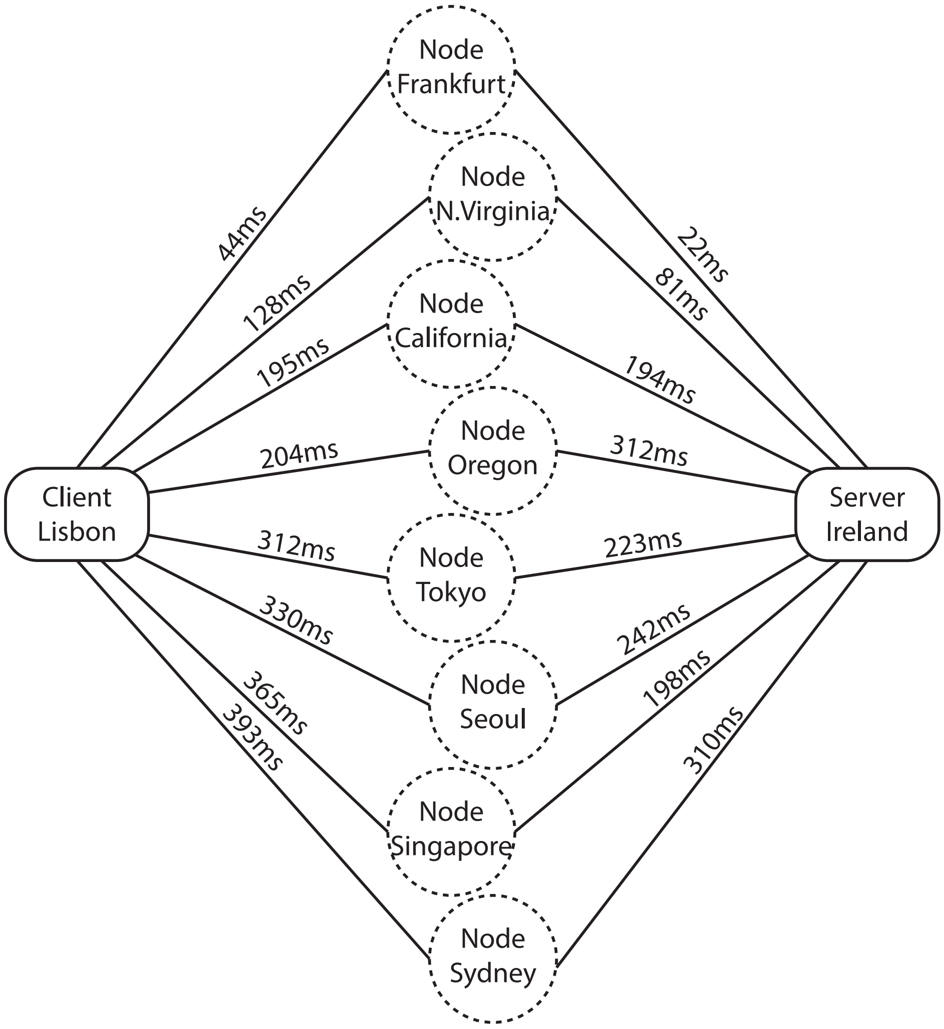


Figure 6.4: Latencies between the Portugal host and the EC2 hosts used for the experimental evaluation.

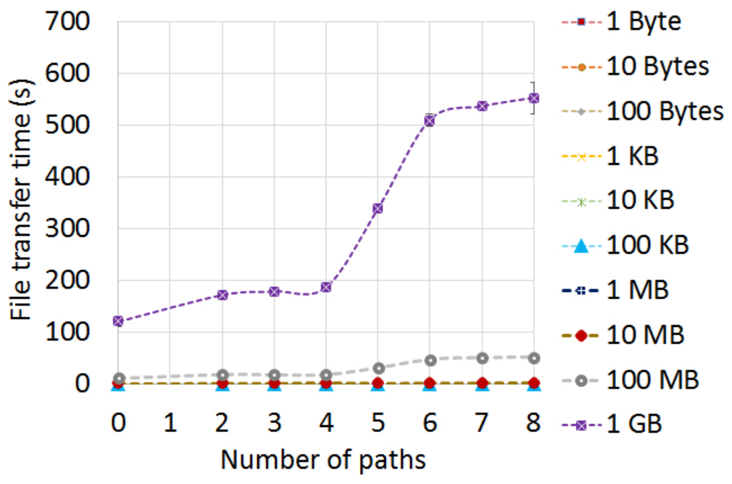


Figure 6.5: Time to transfer a file versus number of paths. 0 paths means a normal TCP connection.

File size	TCP	Fast r.-r.	Strict r.-r.
1 B	49	81	97
10 B	49	95	94
100 B	49	87	98
1 KB	49	94	90
10 KB	49	181	122
100 KB	49	265	201
1 MB	304	382	409
10 MB	1644	2295	3106
100 MB	11556	18836	23452
1 GB	121069	172215	218332

Table 6.3: Average time of sending files using two nodes with two types of round-robin schedulers, compared to a normal TCP connection. All values are presented in milliseconds. Each evaluation was repeated 30 times.

Figure 6.5 shows the values obtained for the time to transfer files of all sizes and from 2 to 8 paths, plus using a standard TCP connection (with no overlay nodes), and includes a 95% confidence intervals, although most are too small to be visible.

The figure shows that splitting the packets in up to four different paths does not generate considerable overhead on the communication. With more than 5 paths, the duration increases due to the overlay nodes that compose the network at that point being farther away from both the source and destination.

6.3.2.2 Performance with diverse paths

Considering the diversity achieved in each of the four regions used on the previous tests, this evaluation considers the two paths with highest diversity, i.e., those with overlay nodes at Frankfurt and California.

Table 6.3, shows the overhead of using these two nodes, in comparison to a normal TCP stream. As shown, there is an overhead of 42% when using the `fast round-robin` scheduler and of 80% when using the `strict round-robin` scheduler, in comparison to TCP. This overhead is the result of sending traffic through a node that is geographically distant from the source and the destination, California. When using a round-robin packet scheduler the whole multi-path connection is conditioned to each path's throughput. In fact, for the `strict round-robin` scheduler, the whole throughput is highly dependant of the path with the smallest bandwidth or highest congestion, since it waits for this channel to have free window space before sending to the next one.

6.3.2.3 Path set up and tear down

Figure 6.6 shows the time for setting up and tearing down the overlay paths. The current Machete implementation is suboptimal in the sense that both the setup and tear down phases are executed sequentially: the device sends the NAT information to a node and waits for its acknowledgment before repeating with the next. According to the location of the node, this time will vary, however, as it can be seen, it always takes longer than one second, but never more than two in our scenario. These two processes will be reimplemented to do these operations in parallel instead of sequentially.

6.3.3 Confidentiality

The usual way of considering confidentiality in the security and cryptography literature is as an absolute property, i.e., a property that is either assured or not. More exactly, in those works, e.g., in protocols like IPsec AH/ESP or TLS, the property is guaranteed as far as no vulnerabilities exist in the protocol design, implementation, and configuration. In this work we do not aim to provide such guarantees but to improve confidentiality in case communication is eavesdropped. Either data is not encrypted or it is but there is a vulnerability. This means that confidentiality was not studied in an absolute perspective, even

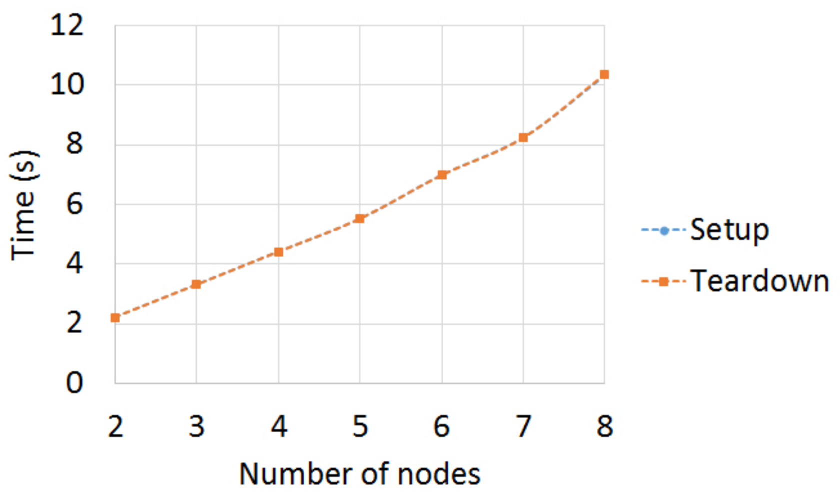
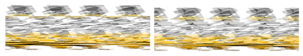


Figure 6.6: Time for setting up and tearing down the overlay paths versus number of overlay nodes used.

(a) The original image sent.


(b) The data sent over the flows is balanced (strict round robin).

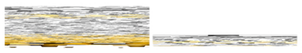

(c) The data sent over the flows is unbalanced (fast round robin).

Figure 6.7: Original image and two reconstructions considering the eavesdropper has access to 2 of the 4 flows in each case.

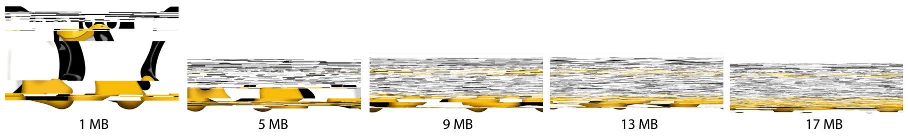


Figure 6.8: Results of capturing the data transferred by the flow with most throughput. The same image was sent with different sizes.

though it is possible that in the second case, it might mitigate the cryptography vulnerabilities providing full confidentiality.

This different way of considering confidentiality led us to use an uncommon way to assess it: a visual evaluation of the reconstruction of an image transmitted over Machete with an eavesdropper with access to one of the flows. For the figures we did the reconstruction assuming the adversary managed to guess the metadata (figure size, color depth, etc.) even if the captured flow did not contain it.

When evaluating the confidentiality that Machete offers it is necessary to remember the operation of MPTCP. The most important factor is the scheduling that is used. MPTCP implements different types of scheduling, however, splitting data in packets in a round-robin fashion is the adequate approach to achieve confidentiality. The multi-path protocol implements two types of round-robin: `fast round-robin`, which takes advantage of the whole throughput of that channel, and `strict round-robin`, that waits for the next channel to have window space before sending the packet. The former is expected to perform faster, but the second to provide access to less data to a potential eavesdropper.

We evaluate two aspects of confidentiality: the effect of the scheduling algorithm, and the effect of the file size.

6.3.3.1 *Effect of the scheduling algorithm*

Figures 6.7b and 6.7c show the different amount of data captured by two of four channels when sending the bitmap picture (the Linux penguin) shown in Figure 6.7a, with both types of round-robin scheduling. The channels shown in each figure are the ones that receive the most distinct amount of data, i.e., the one that receives the most (on the left) and the one that receives the least (on the right).

As shown on Figure 6.7b, using strict round-robin it is possible to notice that both channels receive approximately the same amount of data, resulting of the even distribution of packets. However, a pattern can also be noticed on its reconstruction.

Figure 6.7c shows the results of capturing the data when the `fast round-robin` scheduler is used. As expected, the flow where less data was transferred was the one passing through Sydney's node, the one with lowest throughput. As mentioned before, this scheduler depends on the throughput of each channel when distributing the data, since a channel with better throughput has a larger congestion window to be filled.

In the standard Machete configuration, the result is the one observed in Figure 6.7b, since it does not rely on factors that the user can not control. The performance in the two cases was different, though. By filling the congestion windows with `fast round-robin` scheduling, the file that had 17MB was sent in 4 seconds. By using the strict round-robin the file took 88 seconds to be transferred, which is much slower. The first mode achieves a throughput of 34 Mb/s, whereas the second a mere 1,9 Mb/s.

In short both approaches have their advantages and disadvantages: the first one takes longer and might be susceptible to easier data reconstruction, but provides a good control on how the packets are distributed; the second has its packet distribution dependant of each flows' throughput, but transfers the files faster.

6.3.3.2 Effect of the file size

Another factor to take into account is the size of the files sent. At this point it is important to remember MPTCP's behaviour when creating new flows. The first flow does not wait for the creation of new flows to start transferring data. This means that for very small files (<10KB) MPTCP does not split the packets through any new flows, since this data is sent before any new flow can be established for the stream. Regarding larger files, it is only necessary to experiment with the fast round-robin mode, since the strict round-robin is not influenced by congestion window sizes and, therefore, the sizes are not a factor to take into account.

Figure 6.8 shows the results of capturing the data transferred in the flow with highest throughput (which is the same as mentioned above of 34 Mb/s), when sending the same image with different sizes: from 1 MB to 17 MB. As it can be observed, larger files have stronger resistance to eavesdropping as data is better split among the paths. When the file has 1 MB, a considerable part of the data passes on the highest throughput path and the penguin is somewhat recognizable; with a 17 MB file the contrary is true.

6.4 Summary

Machete is a first effort on providing confidentiality to communications by splitting the packet flows among different physical paths. By establishing dynamic overlay networks, composed by several paths with a single overlay node it was possible to provide physical path diversity. Using MPTCP it was possible to develop a system that transfers data streams (instead of isolated packets) without compromising performance. We evaluated the performance and confidentiality achieved by our implementation, showing that, not only it prevents the attacker from accessing considerable amounts of data, in the case it is trying to spy on the communication, but it also provides different tradeoffs between confidentiality and performance. We believe, that efficiently splitting the communication over physically disjoint channels is the key to maintain confidentiality.

7 Component integration

The security requirements for the SafeCloud middleware were presented in Section 2.3. There, we discussed that, for the attacker to break the confidentiality, privacy or integrity of an SSL/TLS channel, he must: (i) know about the existence of a vulnerability in the channel; and (ii) have access to the endpoints or (iii) have access to the communication.

The SafeCloud middleware components, presented in the previous chapters, provide mechanisms that make it harder for the attacker to achieve (i), (ii), and (iii) individually. To address multiple concerns *at the same time*, we need to *integrate* the existing components.

In this chapter we present examples of the integration of SafeCloud middleware solutions. To address concerns over (i) and (ii), we integrate vtTLS, for a secure channel with cipher diversity, with sKnock, for protected service provisioning. We discuss this integration next, in Section 7.1. To better address concerns over (iii), we integrate Machete, to split communication over different network paths, with Darshana, for detecting possible route interceptions and allow Machete to react to them. We discuss this integration in Section 7.2.

7.1 sKnock integration with vtTLS

vtTLS adds two layers of encryption so that the communication is secure even if a vulnerability is found in one of the layers. sKnock protects the server from scans and allows authorized clients to open and establish connections.

sKnock is a firewall which only allows incoming connections after the clients have successfully authenticated. Therefore any incoming connection is by default rejected until the client opening the connection authenticates with sKnock. This is shown in Figure 7.1.

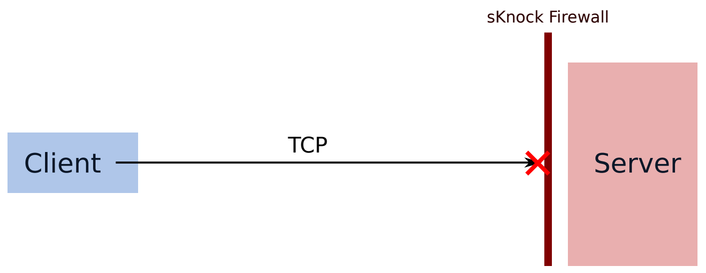
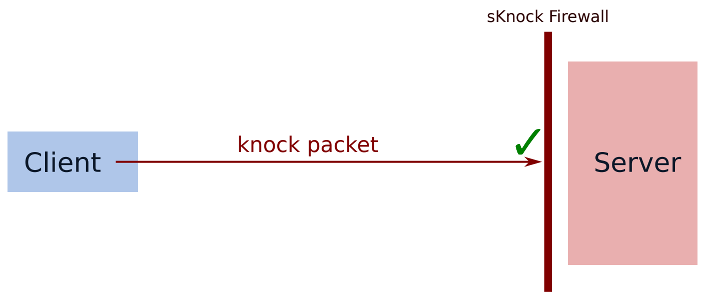


Figure 7.1: sKnock denies unauthenticated incoming connections.

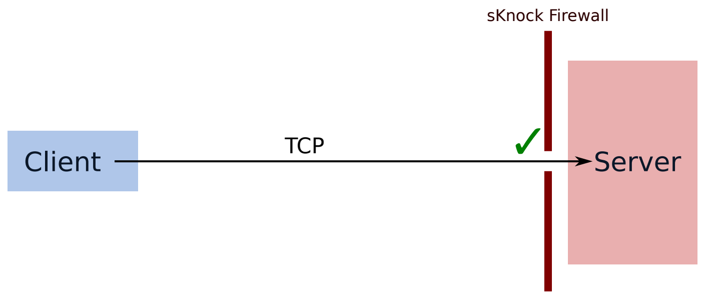
A valid client can authenticate to sKnock by sending a knock packet containing authentication information. If this packet is valid, sKnock opens the firewall for that client to be able to access the authorized services on the server, as shown in Figure 7.2.

After the firewall is opened, the vtTLS client and server negotiate, and then build the multiple layers of protection incrementally, as depicted in Figure 7.3 for 2 levels: first the inner layer is built; then another layer is added over it using a different cipher suite such that the used ciphers are as diverse as possible.

Any other client trying to connect to the server is denied access unless it is authenticated



(a) Client sends a knock packet which is validated by sKnock.



(b) sKnock's Firewall allows incoming connections from the client.

Figure 7.2: sKnock allows authenticated clients which send valid knock packets before opening connections to services on the server.

using sKnock first.

Once the first client's vTTLS channel is closed, the firewall closes the access to that client also. If the same client wants to access the service again, it needs to reauthenticate to sKnock.

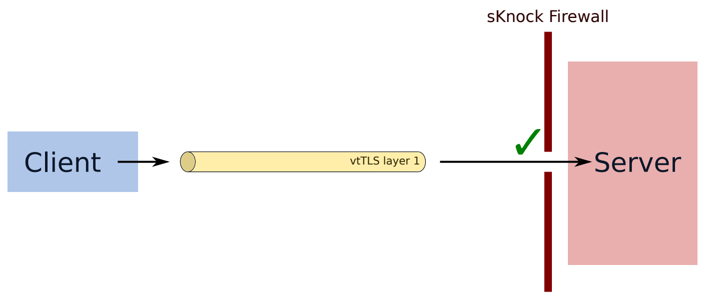
7.2 Darshana integration with Machete

Machete and Darshana are both route-aware communication solutions. Machete allows the use of multiple communication paths for added security, and Darshana is able to monitor network connections and detect possible route deviations (*hijacks*). Figure 7.4 presents the integrated architecture for Machete and Darshana.

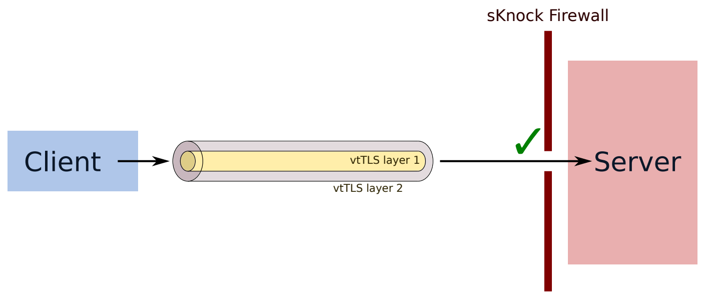
To perform the monitoring of the multiple paths, several Darshana instances have to be deployed: one instance for each path from Sender to Overlay Node; one instance for each path from Overlay Node to Receiver.

All instances report to a single DAR (Darshana Alert Receiver), running in the Sender. In case any Darshana instance detects a path anomaly – that can be interpreted as a route hijack – it reports back to the DAR. It is now up to Machete to decide what to do when an alert is received. The default behavior is to stop using the possibly compromised path.

The alerts are sent using UDP communication. A Darshana Alert packet format was defined, containing: timestamp, type of alert, and details for the metrics that triggered the alarm.



(a) Client builds the inner vtTLS protection layer.



(b) Client builds the outer vtTLS protection layer.

Figure 7.3: vtTLS allows multiple layers of protection for communication data.

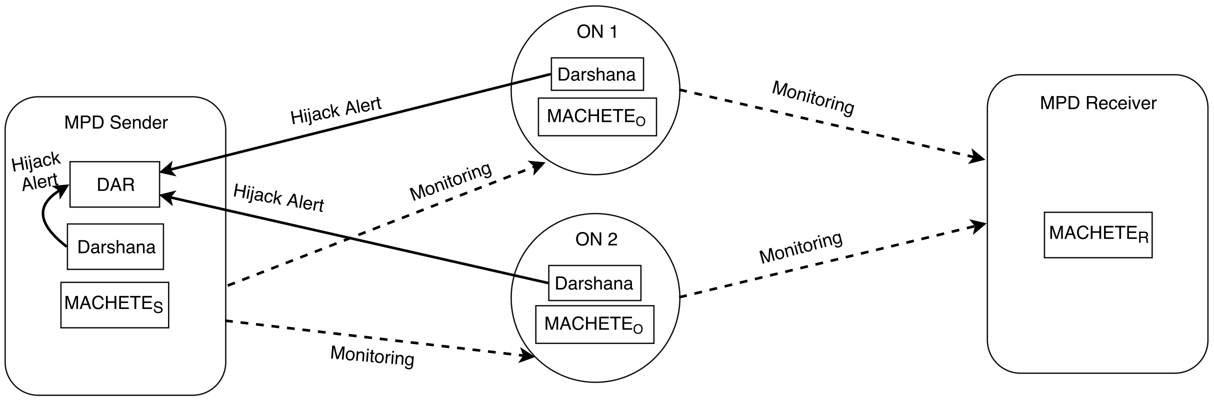


Figure 7.4: Machete and Darshana integration. DAR is the Darshana Alert Receiver. S stands for Sender, ON for Overlay Node, and R for Receiver.

7.3 Summary

In this chapter we presented specific examples of how the developed secure SafeCloud middleware communication solutions can be combined to address multiple security concerns at the same time. By combining vtTLS and sKnock both endpoints can be protected, and the data in transit is also further protected. By combining Machete and Darshana, the multiple path communication is monitored and, in the event of a likely hijack, the communication through the affected path can be stopped. These are just two examples of the added protection capabilities made possible by the SafeCloud middleware components.

8 Conclusion

This document presents the final implementation of the SafeCloud private communication middleware components. These components, as a whole, aim to provide confidentiality, integrity, and authenticity – plus availability - as other secure channels like SSL/TLS, IPsec; but assuming powerful adversaries that may be able to break some of the assumptions that make existing channels secure e.g., that a certain cryptographic algorithm is secure.

The deliverable presents:

- The business requirements for the middleware components;
- Details about the latest version of each component: vulnerability-tolerant channels (vtTLS), protected service provisioning (sKnock), route monitoring (Darshana), and multi-path communication (Machete);
- Presentation of middleware component integrations, with details on how added-value is derived from the integration.

The final version of the middleware components will be applied in the use cases of the project, as necessary according to their business requirements.

Bibliography

[ABD$^+$15] 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.

[ABKM01] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 131–145, 2001.

[AC77] 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.

[ACO$^+$06] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clémence Magnien, and Renata Teixeira. Avoiding traceroute anomalies with Paris traceroute. *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 153–158, 2006.

[AD03] Y. Amir and C. Danilov. Reliable communication in overlay networks. *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 511–520, 2003.

[ADHP16a] 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.

[ADHP16b] Martin R. Albrecht, Jean Paul Degabriele, Torben Brandt Hansen, and Kenneth 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.

[AJ$^+$05] John Aycock, M Jacobson, et al. Improved port knocking with strong authentication. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10–pp. IEEE, 2005.

[Ama] Amazon Web Services, https://aws.amazon.com/.

[AMS$^+$03] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based analysis of multihoming. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 353–364, 2003.

[AMSS08] A. Akella, B. Maggs, S. Seshan, and A. Shaikh. On the Performance Benefits of Multihoming Route Control. *IEEE/ACM Transactions on Networking*, 16(1):91–104, 2008.

[BACF08] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 163–176, April 2008.

[BBB$^+$07] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Nist special publication 800-57. *NIST Special Publication*, 800(57):1–142, 2007.

[BD12] 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.

[bel] The new threat: Targeted Internet traffic misdirection, http://research.dyn.com/2013/11/mitm-internet-hijacking/.

[BFM13] L. Boccassi, M. M. Fayed, and M. K. Marina. Binder: a system to aggregate multiple Internet gateways in community networks. In *Proceedings of the 2013 ACM MobiCom Workshop on Lowest Cost Denominator Networking for Universal Access*, pages 3–8, 2013.

[BFMR10] Kevin Butler, Toni Farley, Patrick McDaniel, and Jennifer Rexford. A survey of BGP security issues and solutions. *Proceedings of the IEEE*, 98(1):100–122, 2010.

[BFZ07] H. Ballani, P. Francis, and X. Zhang. A study of prefix hijacking and interception in the Internet. *ACM SIGCOMM Computer Communication*, 2007.

[BKR11] 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.

[BL13] Daniel J Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. *URL: http://safecurves. cr. yp. to*, 2013.

[BL16] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467, 2016.

[Bon14] O. Bonaventure. MPTLS: Making TLS and multipath TCP stronger together. *IETF, Individual Submission, Internet Draft draft-bonaventure-mptcp-tls-00*, 2014.

[BPB11] S. Barré, C. Paasch, and O. Bonaventure. Multipath TCP: from theory to practice. In *Networking 2011*, pages 444–457. Springer, 2011.

[Bro08] Martin Brown. Pakistan hijacks youtube video, http://research.dyn.com/-2008/02/pakistan-hijacks-youtube-1/, 2008.

[Car14] Ricardo Carvalho. Authentication security through diversity and redundancy for cloud computing. Master's thesis, Instituto Superior Técnico, Lisbon, Portugal, 2014.

[CDFW14] M. Carvalho, J. DeMott, R. Ford, and D. Wheeler. Heartbleed 101. *IEEE Security & Privacy*, 12(4):63–67, 2014.

[CF14] M. Carvalho and R. Ford. Moving-target defenses for computer networks. *IEEE Security and Privacy*, 12(2):73–76, 2014.

[CGL$^+$14] Y. Cheng, M. T. Gardner, J. Li, R. May, D. Medhi, and J. P. G. Sterbenz. Optimised heuristics for a geodiverse routing protocol. In *2014 10th International Conference on the Design of Reliable Communication Networks (DRCN)*, pages 1–9, April 2014.

[Che16] Roger Cheng. Tech industry rallies around Apple in its iPhone fight with FBI. CNET. https://www.cnet.com/news/iphone-fight-against-fbi-tech-groups-industry-leaders-throw-support-behind-apple/, March 2016.

[CK15] Eric F Crist and Jan Just Keijser. *Mastering OpenVPN*. Packt Publishing Ltd, 2015.

[Clo11] Cloud Security Alliance. Security guidance for critical areas of focus v3.0. https://downloads.cloudsecurityalliance.org/assets/research/security-guidance/csaguide.v3.0.pdf, 2011.

[Cow10] Jim Cowie. China 18-Minute Mystery, http://research.dyn.com/2010/-11/chinas-18-minute-mystery/, 2010.

[cym] Team Cymru, IP to ASN mapping, http://www.team-cymru.org/IP-ASN-mapping.html, url = http://www.team-cymru.org/IP-ASN-mapping.html.

[DBVV11] J. Díez, M. Bagnulo, F. Valera, and I. Vidal. Security for multipath TCP: a constructive approach. *International Journal of Internet Protocol Technology*, 6(3):146–155, 2011.

[dep] DepSpace, https://github.com/bft-smart/depspace.

[DR08] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol, version 1.2, IETF RFC 5246, 2008.

[ENI14] ENISA. Algorithms, key size and parameters report – 2014. nov 2014.

[ENTK11] D. Evans, A. Nguyen-Tuong, and J. Knight. Effectiveness of moving target defenses. In *Moving Target Defense*, volume 54, pages 29–48. Springer, 2011.

[Fed16] Federal Communications Commission. Cyber security planning guide. https://www.fcc.gov/cyber/cyberplanner.pdf, 2016.

[FKL$^+$01] 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.

[FRH$^+$11] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural guidelines for multipath TCP development, IETF RFC 6182, 2011.

[FWC16] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL implementation of ECDSA with a few signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1505–1515, 2016.

[GBG$^+$11] 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.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, January 1985.

[GGCS02] Vipul Gupta, Sumit Gupta, Sheueling Chang, and Douglas Stebila. Performance analysis of elliptic curve cryptography for SSL. In *Proceedings of the 1st ACM Workshop on Wireless Security*, pages 87–94. ACM, 2002.

[Gol16] Rafi Goldberg. Lack of trust in internet privacy and security may deter economic and other online activities. National Telecommunications and Information Administration, United States Department of Commerce. https://www.ntia.doc.gov/blog/2016/lack-trust-internet-privacy-and-security-may-deter-economic-and-other-online-activities, May 2016.

[Goo15] Dan Goodin. Hacking Team orchestrated brazen BGP hack to hijack IPs it did not own. 2015.

[Hau16] Laura Hautala. The Snowden effect: Privacy is good for business. CNET. https://www.cnet.com/news/the-snowden-effect-privacy-is-good-for-business-nsa-data-collection/, June 2016.

[HMM07] Xin Hu, Mao, and Z. Morley. Accurate Real-time Identification of IP Prefix Hijacking. *IEEE Symposium on Security and Privacy*, (2):3–17, 2007.

[HNC18] F. Helfert, H. Niedermayer, and G. Carle. Evaluation of Algorithms for Multipath Route Selection over the Internet. In *Proceedings of the Conference on Design of Reliable Communication Networks*, 2018.

[HNL$^+$13] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 1–11, 2013.

[HR08] J. He and J. Rexford. Toward Internet-wide multipath routing. *IEEE Network*, 22(2):16–21, 2008.

[IT12] ITU-T. Privacy in cloud computing. http://www.itu.int/oth/T2301000016/en, March 2012.

[ITU05] ITU ITU. Information technology–open systems interconnection–the directory: Publickey and attribute certificate frameworks. *Suíça, Genebra, ago*, 2005.

[JKBM$^+$08] J.P. John, E. Katz-Bessett, H.V. Madhyastha, A. Krishnamurthy, D.Wetherall, and T. Anderson. Studying blackholes in the Internet with hubble. *Proceedings of NSDI*, 2008.

[KAF$^+$10] 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.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication, IETF RFC 2104, 1997.

[KFR06] Josh Karlin, Stephanie Forrest, and Jennifer Rexford. Pretty good BGP: improving BGP by cautiously adopting routes. *Proceedings of the 2006 IEEE International Conference on Network Protocols*, pages 290–299, 2006.

[KG14] Julian Kirsch and C Grothoff. Knock: Practical and secure stealthy servers, 2014.

[KK11] S. Kanno and M. Kanda. Addition of the camellia cipher suites to transport layer security (tls), IETF RFC 6367, 2011.

[KKJ01] J.-S. Kim, K. Kim, and S.-I. Jung. Building a high-performance communication layer over virtual interface architecture on Linux clusters. In *Proceedings of the 15th ACM International Conference on Supercomputing*, pages 335–347, 2001.

[KPW06] K.Butler, P.McDaniel, and W.Aiello. Optimizing BGP security by exploiting path stability. *Proceedings ACM Conference on Computer and Communications Security*, 2006.

[Krz03] Martin Krzywinski. Port knocking from the inside out. *SysAdmin Magazine*, 12(6):12–17, 2003.

[KS05] S. Kent and K. Seo. Security architecture for the internet protocol, IETF RFC 4301, 2005.

[Lep16] M. Lepinski. BGPSEC protocol specification, draft-ietf-sidr-bgpsec-protocol-17. 2016.

[lft] Layer Four Traceroute (LFT) Project, http://pwhois.org/lft/index.who.

[LHBF14] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 276–291, 2014.

[LHH08] Matthew Luckie, Young Hyun, and Bradley Huffaker. Traceroute probe method and forward IP path inference. *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement conference*, page 311, 2008.

[LLK13] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency, IETF RFC 6962, 2013.

[LLV$^+$15] Dave Levin, Youndo Lee, Luke Valenta, Zhihao Li, Victoria Lai, Cristian Lumezanu, Neil Spring, and Bobby Bhattacharjee. Alibi routing. *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 611–624, 2015.

[LMPW06] Mohit Lad, D Massey, D Pei, and Y Wu. PHAS: A prefix hijack alert system. *Proceedings of the Usenix Security Conference*, pages 153–166, 2006.

[LS04] 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.

[max] MaxMind Developer Documentation, http://dev.maxmind.com/.

[Mer79] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems.* PhD thesis, Stanford, CA, USA, 1979.

[Mic15] Microsoft. Trusted cloud: Microsoft Azure security, privacy, and compliance. http://download.microsoft.com/download/1/6/0/160216AA-8445-480B-B60F-5C8EC8067FCA/WindowsAzure-SecurityPrivacyCompliance.pdf, 2015.

[MM05] M. Menth and R. Martin. Network resilience through multi-topology routing. In *DRCN 2005). Proceedings.5th International Workshop on Design of Reliable Communication Networks, 2005.*, pages 271–277, Oct 2005.

[MMETC05] Antonio Izquierdo Manzanares, Joaquín Torres Márquez, Juan M Estevez-Tapiador, and Julio César Hernández Castro. Attacks on port knocking authentication mechanism. In *International Conference on Computational Science and Its Applications*, pages 1292–1300. Springer, 2005.

[MMPR11] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-based one-time password algorithm, IETF RFC 6238, 2011.

[MvOV96] A. Menezes, P. van Oorschot, and S. Vanstone. Public-Key Encryption. In *Handbook of Applied Cryptography*, chapter 8. CRC Press, Inc., 1996.

[opt12] ARRL. The Handbook for Radio Communications. The ARRL, 89th edition. 2012.

[PB$^+$] C. Paasch, S. Barre, et al. Multipath TCP in the Linux Kernel, http://www.multipath-tcp.org.

[PCVB13] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush. From Paris to Tokyo: On the suitability of ping to measure latency. In *Proceedings of the 2013 ACM Internet Measurement Conference*, pages 427–432, 2013.

[Pew14] Pew Research Center. Public perceptions of privacy and security in the post-Snowden era. http://www.pewinternet.org/2014/11/12/public-privacy-perceptions/, November 2014.

[pla] PlanetLab: global research networks that support the development of new network services, https://www.planet-lab.eu/.

[RD01] V. Rijmen and J. Daemen. Advanced Encryption Standard. *U.S. National Institute of Standards and Technology (NIST)*, 2009:8–12, 2001.

[RL95] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4), IETF RFC 1771, 1995.

[RS10] Tom Roeder and Fred B. Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems*, 28:4:1–4:54, July 2010.

[Rus02] R. Russell. Linux 2.4 packet filtering howto, revision 1.26, January 2002.

[SBC$^+$10] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, April 2010.

[SBK$^+$17] 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.

[SCK00] S.Kent, C.Lynn, and K.Seo. Secure border gateway protocol (S-BGP). *IEEE JSAC Special Issue on Network Security*, 2000.

[Sho95] 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.

[SHSA15] Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing known attacks on transport layer security (TLS) and datagram tls (DTLS), IETF RFC 7457, 2015.

[Sim95] William Simpson. IP in IP tunneling, IETF RFC 1853, 1995.

[SKP15] M. Stevens, P. Karpman, and T. Peyrin. Freestart collision on full SHA-1. Cryptology ePrint Archive, Report 2015/967, 2015.

[Som16] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504, 2016.

[Ste12] M. Stevens. *Attacks on Hash Functions and Applications*. PhD thesis, Mathematical Institute, Leiden University, 2012.

[STW12] R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension (RFC 6520), 2012.

[Suu74] J. W. Suurballe. Disjoint paths in a network. *Networks*, 4(2):125–145, 1974.

[SXW$^+$12] Xingang Shi, Yang Xiang, Zhiliang Wang, Xia Yin, and Jianping Wu. Detecting prefix hijackings in the Internet with ARGUS. *Proceedings of the 2012 ACM Internet Measurement Conference*, 2012.

[VHT09] Eugene Y Vasserman, Nicholas Hopper, and James Tyra. Silentknock: practical, provably undetectable authentication. *International Journal of Information Security*, 8(2):121–135, 2009.

[VMC02] John Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL: Cryptography for Secure Communications*. O'Reilly, 2002.

[VNC03] 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.

[Vod14] Vodafone. Sustainability report 2013/14. https://www.vodafone.com/ content/dam/sustainability/2014/pdf/vodafone_full_report_2014.pdf, 2014.

[Wei05] Hans Weibel. High precision clock synchronization according to ieee 1588 implementation and performance issues. *Proc. Embedded World 2005*, 2005.

[WJP03] W.Aiello, J.Ioannidis, and P.McDaniel. Origin authentication in interdomain routing. *Proceedings of ACM Conference on Computer and Communications Security*, 2003.

[WR11] D. Wischik and C. Raiciu. Design, implementation and evaluation of congestion control for multipath TCP. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 99–112, 2011.

[WYY05] 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.

[XKI03] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 260–269, 2003.

[YL06] T. Ylonen and C. Lonvick. The secure shell (SSH) protocol architecture, IETF RFC 4251, 2006.

[YR06] S. Hares Y. Rekhter, T. Li. A border gateway protocol 4, IETF RFC 4271, 2006.

[ZDJC09] Min Zhang, Maurizio Dusi, Wolfgang John, and Changjia Chen. Analysis of udp traffic usage on internet backbone links. In *Applications and the Internet, 2009. SAINT'09. Ninth Annual International Symposium on*, pages 280–281. IEEE, 2009.

[ZJP$^+$07] Changxi Zheng, Lusheng Ji, Dan Pei, Jia Wang, and Paul Francis. A light-weight distributed scheme for detecting IP prefix hijacks in real-time. *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 277–288, 2007.

[ZPW$^+$01] Xiaoliang Zhao, Dan Pei, Lan Wang, Dan Massey, Allison Mankin, S. Felix Wu, and Lixia Zhang. An analysis of BGP multiple origin AS (MOAS) conflicts. *Proceedings of the First ACM SIGCOMM Workshop on Internet Measurement Workshop*, pages 31–35, 2001.

[ZZH$^+$08] Zheng Zhang, Ying Zhang, Y. Charlie Hu, Z. Morley Mao, and Randy Bush. iSPY: detecting IP prefix hijacking on my own. *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, 2008.