Private communication middleware architecture

D1.1

Project reference no. 653884

February 2016

European Commission

Horizon 2020
European Union funding
for Research & Innovation

Document information

Scheduled delivery	01.03.2016
Actual delivery	01.03.2016
Version	1.0
Responsible Partner	INESC-ID

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
23.12.2015	M. Correia	Draft	0.1	Initial version
04.01.2016	S. Totakura	Draft	0.2	Revised version
17.02.2016	M. Pardal	Draft	0.3	Revised version
26.02.2016	M. Pardal	Final	1.0	Reviewers comments incorporated.

Contributors

Miguel Pardal (INESC-ID)
Miguel Correia (INESC-ID)
Sree Harsha Totakura (TUM)
Georg Carle (TUM)
Karan Balu (INESC-ID)
André Joaquim (INESC-ID)
Diogo Raposo (INESC-ID)

Internal reviewers

V. Schiavoni (UniNE)
K. Tarbe (CYBER)

Acknowledgements

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

Glossary of acronyms

Acronym	Definition
API	Application Programming Interface
AS	Autonomous System
BSD	Berkeley Sockets Distribution
BGP	Border Gateway Protocol
CA	Certification Authority
CBC	Cipher Block Chaining
CN	Certificate Notary
CRL	Certification Revocation List
DES	Data Encryption Standard
DHE	Diffie-Hellman
ENISA	European Network and Information Security Agency
HMAC	Hashed Message Authentication Code
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPsec	IP Security
IV	Initialization Vector
MAC	Message Authentication Code
MD	Message Digest
NIST	National Institute of Standards and Technology
MPTCP	Multi-Path TCP
OSI	Open Systems Interconnection
PKI	Public-Key Infrastructure
RFC	Request for Comments
RIR	Regional Internet Registrars
RMS	Relay Membership Service
SCCA	SafeCloud Client Agent
SCCL	SafeCloud Client Library
SCRWP	SafeCloud Reverse Web Proxy
SCSA	SafeCloud Server Agent
SCSL	SafeCloud Server Library
SCWP	SafeCloud Web Proxy
SCSV	Signaling Cipher Suite Value (SCSV)
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
SSH	Secure Shell
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time To Live
UDP	User Datagram Protocol

Table of contents

Executive summary

Data communication is an important source of concerns about privacy and confidentiality. For example, in recent months alarms were raised about: weaknesses in cryptographic algorithms used to encrypt communications, attacks against the algorithm used to agree upon shared keys in TLS, the deviation of huge chunks of Internet traffic to far away countries, and the espionage of traffic inside the networks of major cloud computing providers. All these issues are related to communication insecurity and privacy leaks.

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 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. An 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].

The present deliverable (D1.1) is the first of WP1. It presents a *preliminary design of the secure communication middleware:* the threats it assumes, the service it will provide, and its architecture.

The middleware is concerned with two forms of communication: *machine-to-cloud* and *cloud-to-cloud*. SafeCloud does not envisage the need to protect data privacy in multicast, anycast or broadcast communications, so the middleware provides only *unicast communication*, i.e., communication between two *endpoints*. These endpoints can be user terminals, computers in the clouds, among others. Communication is connection-oriented, similarly to protocols like TCP or SSL, as we also do not envisage the need to support datagram communication (e.g., as UDP).

The middleware is implemented at the application layer of the OSI model and of the Internet protocol stack, following the end-to-end argument (most complexity should be implemented at the higher layers of the protocol stack) and practical considerations about the difficulty of implementing mechanisms at lower layers in the Internet (from the network layer down changes may be needed in equipment of Internet service providers, which is unpractical).

This deliverable is organized as follows.

- Section 2 presents the threats that the middleware has to tackle, i.e., the problems it has to solve or at least mitigate.
- Section 3 presents the services that the middleware will provide, explaining how they address the threats of Section 2.
- Section 4 presents the architecture of the middleware.
- Finally, Section 5 concludes the report.

2 Threats

As mentioned in the previous section, we aim to provide security despite *powerful adversaries* that may be able to break some of the usual assumptions made by secure channel solutions such as SSL/TLS, IPsec, and SSH. Specifically, *we consider four threats,* which we explain in the following sections:

- **Secure channel component vulnerabilities**: secure channels (e.g., TLS, IPsec) are based on components that may be vulnerable (e.g., RC4, MD5, SHA-1);
- **Service identification**: servers that offer services via well-known ports are vulnerable to attacks that identify services (port scanning/fingerprinting);
- **Man-in-the-middle attacks**: an attacker may impersonate an endpoint identity and thereby defeat security mechanisms that keep the communication to the endpoint secure;
- **Route hijacking**: traffic may be deviated and then eavesdropped.

We consider as main example of secure channel protocol the widely adopted *Transport Layer Security* (TLS) [DR08]. 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 [FKC11]. Defined in 1999, TLS did not introduce major changes to SSL, but they were enough to make TLS 1.0 incompatible with SSL 3.0. In order to grant compatibility, a TLS 1.0 connection can be downgraded to SSL 3.0, bringing security issues (cf. Section 2.1.4). TLS 1.1 and TLS 1.2 are upgrades that brought some improvements such as mitigating CBC (cipher block chaining) attacks and supporting more block cipher modes of operation to use with AES. TLS is divided into two sub-protocols, Handshake and Record, constituted by several mechanisms each. The Handshake protocol is used to establish or re-establish communication between a server and a client. The Record protocol is used to protect the messages sent and received.

2.1 Secure channel component vulnerabilities

This section provides an argument that secure channels may be vulnerable because their components may contain vulnerabilities (Sections 2.1.1-2.1.3) or the protocol itself may be vulnerable (Section 2.1.4).

2.1.1 Vulnerabilities in asymmetric cipher mechanisms

Proposed by Rivest *et al.* in 1978, RSA is a cryptographic mechanism used to cipher and sign messages or data [RSA78]. RSA's security is based on two problems: factorization of large integers and the RSA problem (the task of performing an RSA private-key operation given only the public key) [MOV96]. RSA's strength is inversely proportional to the available computational power. As the years pass, it is expected that performing the factorization of large integers becomes a feasible task. RSA will be broken when those problems can be solved within a practical amount of time.

Kleinjung *et al.* performed the factorization of RSA-768, a RSA number with 232 digits [KAF+10]. The researchers state that they spent almost two years in the whole process, which is clearly a non-feasible time. Factorizing a large integer is a very different concept from breaking RSA. RSA is still secure. As of 2010, the researchers concluded

that RSA-1024 would be factored within five years, i.e., in 2015. As for now, no factorization of RSA-1024 has been publicly announced.

Shor invented an algorithm to factorize integers in polynomial time using quantum computing [S95]. However, for the time being quantum computers able to do such factorization are still theoretical.

Other attacks, unrelated to the problems mentioned above, can target RSA. A chosen plaintext attack [DH76] exploits the fact that RSA is deterministic – given the same input, the same output will be produced. Let us assume the attacker has the public key, knows some of the content of the original message, and has the original message ciphered. He can try to cipher multiple plaintexts of his own with the public key, compare them with the original ciphered message, and discover the message.

A chosen ciphertext attack [RS92] is another possible attack to RSA. It benefits from the multiplicative property of RSA. The product of two ciphertexts is equal to ciphering the product of the two plaintexts. In these attacks, an attacker sends a ciphered message of his own, consisting on the original ciphered message M multiplied by a random value *R*, ciphered with the same key as *M*. If the attack is successful, the attacker can now obtain *M* by decomposing the message sent. These attacks assume that an attacker can obtain plaintexts from ciphertexts from the system. Adaptive chosen ciphertext attacks are an adaptive form of chosen ciphertext attacks in the sense that they use the results of previously sent ciphertexts to select future ciphertexts.

2.1.2 Vulnerabilities in symmetric cipher mechanisms

Triple DES, or 3DES, is an encryption mechanism based on DES [F99]. As the name suggests, Triple DES consists on multiple encryption using DES. Triple DES can be implemented using one of three keying options:

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

Associated with each keying option there is a set of vulnerabilities. Keying option i uses three different 56-bit keys, i.e., a 168-bit key, but due to meet-in-the-middle attacks, the real security is 112-bit. Lucks presented an attack to keying option i of Triple DES [L98]. The proposed attack consists in a meet-in-the-middle attack that reduced the number of steps needed to roughly 2^{108} steps from 2^{112} steps. While being an improvement compared to the brute force attack, this attack still requires a lot of time to be performed, which makes it impracticable. NIST states that Triple DES with keying option i is secure until the end of 2030. From there onwards, its use is disallowed [NIST12].

Keying option ii uses two different 56-bit keys, i.e., a 112-bit key. Keying option ii, also known as two-key triple encryption, can be attacked using chosen plaintext attacks with about 2^k steps with $k = 56$ [MH81]. Merkle *et al.* concluded it is preferable to use a single encryption algorithm with a longer key rather than a multiple encryption algorithm with a smaller key.

Keying option iii has the same security as DES, which makes it an insecure option.

2.1.3 Vulnerabilities in hash functions

A hash function is a function that outputs a hash given a byte array as input. The main applications of hash functions in modern cryptography regard data integrity and message authentication. Sometimes also called message digest or digital fingerprint, the hash is a compact representation of the input array and can be used to identify it [MOV96].

According to Menezes *et al.*, a hash function *h* must have at least two properties [MOV96]:

- *Compression* – *h* maps an input array *s* of finite length to an output h(s) of a fixed length *x*;
- *Ease of computation* – given *h* and an input array *s*, the hash h(s) is easy to compute.

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

- *Preimage resistance* – for all pre-specified outputs, it is computationally infeasible to find any input array which hashes to that output, i.e., finding the input array s, given the output h(s);
- *Second preimage resistance* – it is computationally infeasible to find any second input array which has the same hash as any specified input array, i.e., given s1, finding s2 <>s1, where h(s1) = h(s2);
- *Collision resistance* – it is computationally infeasible to find two input arrays whose hash is identical, i.e., finding s1 != s2, where h(s1) = h(s2).

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

MD5 is a cryptographic hash function that, although being proved to be insecure, is still widely used nowadays. MD5 supersedes MD4, and was created by Rivest in 1991 [R92]. MD5 produces a 128-bit message digest and is commonly used to verify data integrity. Wang et al. proved that MD5 is not collision-resistant [WY05]. The employed attack was a differential attack that is a form of attack that consists in studying how differences in the input affect the output.

The Secure Hash Algorithm 1 (SHA-1) is another cryptographic hash function. It produces a 160-bit message digest. Although there have not been publicly found actual collisions for SHA-1, it is considered insecure and the use of SHA-2 or SHA-3 is recommended [S15]. Other attacks have been successful 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 [M79] 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 Initialization Vector (IV). Even though there are successful freestart collision attacks on SHA1 it does not imply that SHA1 itself is insecure.

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} [WY05]. 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 $2^{65.3}$ [St12].

Nowadays it is still computationally expensive to perform these number of calculations. It is expected that by the year of 2021, a collision attack will be affordable [Ss12].

2.1.4 SSL/TLS vulnerabilities

This section concludes by showing that the SSL/TLS protocol itself may be vulnerable.

TLS vulnerabilities can be classified in two types: specification vulnerabilities and implementation vulnerabilities. Specification vulnerabilities concern the protocol itself. A *specification vulnerability* can only be fixed by a new protocol version or an extension. We may argue that the deprecation of a cryptographic mechanism is a protocol's design flaw as the protocol should not support an insecure mechanism. *Implementation vulnerabilities* are related to vulnerabilities in certain implementations of SSL/TLS, such as OpenSSL, or browsers' implementations. The Internet Engineering Task Force (IETF) released RFC 7457 [SHS15] on February 2015 containing the summary of known attacks to TLS specifications and TLS implementations. In this section we present the most recent (known) attacks to TLS.

Starting with specification vulnerabilities, Logjam is the most recent attack and it was presented in May 2015 [ABD+15]. The Logjam attack consists in exploiting Diffie-Hellman key exchange weaknesses. Logjam is a man-in-the-middle attack that downgrades the connection to a weakened Diffie-Hellman mode. This Diffie-Hellman with weak parameters is one attack made possible by the restrictions imposed by the U.S.A. export of cryptography. In the 1990's, the United States of America legislated some restrictions over exporting cryptography. In order to support SSL/TLS in some countries not allowed to import U.S.A. cryptography, SSL/TLS supports weakened Diffie-Hellman modes that are called the EXPORT modes [V15].

Previous attacks, such as the one made possible due to the FREAK vulnerability [BBD+15] have already made use of this weakness. Adrian *et al.* consider Logjam a result of a protocol specification vulnerability due to the fact of TLS still allowing the use of Diffie-Hellman with weak parameters [ABD+15].

In order to understand the attack itself, we introduce the concept of number field sieve algorithm. A number field sieve algorithm is an efficient algorithm used to factor integers bigger than one hundred digits. The authors used a number field sieve algorithm to precompute two weak 512-bit Diffie-Hellman groups used by more than 92% of the vulnerable servers parameters [ABD+15]. This approach was taken due to the fact that it is computationally heavy to generate prime numbers with the desired characteristics.

The Logjam man-in-the-middle attack changes the current cipher suites to DHE_EXPORT (which provides EXPORT modes), forcing the use of weakened Diffie-Hellman key exchange parameters. As the server supports DHE_EXPORT, a completely valid Diffie-Hellman mode, the handshake proceeds without the server noticing the attack. The server proceeds to compute its premaster secret using weakened Diffie-Hellman parameters. From the client's point-of-view, the server chose a seemingly normal

ephemeral Diffie-Hellman (DHE) option and proceeds to compute its secret also with weak Diffie-Hellman parameters. By this point, the man-in-the-middle can use the precomputation results to break one of the secrets and establish the connection to the client pretending to be the server. One aspect worth noticing is that this attack will only succeed if the server does not refuse to accept DHE_EXPORT mode.

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

Another attack concerning the protocol specification is a padding attack named POODLE. POODLE stands for Padding Oracle On Downgraded Legacy Encryption and was presented by Google engineers in 2014 [MDK14]. The origin of this attack is the backward compatibility with SSL 3.0 of many TLS implementations. To be successful, the attacker should induce a downgrade attack first, in order to transition from TLS 1.x to SSL 3.0.

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

The POODLE attack can be used to decrypt HTTP cookies in web sites. The authors state that for every byte revealed, 256 SSL 3.0 requests are needed. The attack has been proved to be possible in TLS [L14]. The implementations affected are those that do not properly check the padding used.

The most obvious solution is to avoid SSL 3.0 by disallowing the backwards compatibility and deprecating SSL 3.0. Another solution is to use TLS Fallback SCSV, as described in [ML14]. This signaling cipher suite value (SCSV) intends to prevent unnecessary downgrade of the connection when both client and server actually do support the most recent version of TLS.

Regarding TLS implementation vulnerabilities, Heartbleed, discovered in 2014, is one of the most recent ones (CVE-2014-0160). Its name comes from the mechanism where the vulnerability lies, the heartbeat extension. Heartbleed was a security vulnerability in OpenSSL 1.0.1, through 1.0.1f, when the heartbeat extension [STW12] was introduced and enabled by default. The Heartbleed vulnerability allowed an attacker to perform a buffer over-read [CDF+14]. A buffer over-read happens when more data is read than allowed. This can be used to access the contents of other, possibly sensible, program variables.

2.2 Service identification

Almost all of the cyberattacks are performed after identifying the vulnerabilities of the targeted system. These are found out by acquiring knowledge about the type of operating system the system is running, and the type of services running on it. Often, the vulnerabilities are a result of a programming error in services, which result in a buffer-overflow or a similar exploit, or as a result of misconfiguration of services. Either way, knowledge about the running services helps an attacker to determine the attack vector

to be used. If this information is hidden to the attacker, its attack vector is reduced and thus the attacks hindered.

A common and easy way of hiding information about the services running on the server is to run them on non-standard port numbers. This helps to evade port scans by an attacker over commonly used port numbers. However, should the attacker conduct a scan over all ports or by sheer luck learn a non-standard port being open, he would be able to determine the type and in some cases the version of service by connecting to the service and observing how the service responds. This process is called banner grabbing or service fingerprinting.

A defense against service fingerprinting is to use a Firewall or an Intrusion Detection System (IDS) to observe and block port scans. However, this could be defeated if the attacker has enough patience to temporally spread out the scans over a long period of time or has enough IP addresses to conduct the scans (as in the case with a botnet).

2.3 Man-in-the-middle attacks

In these attacks an attacker places himself in between the communicating entities and tries to impersonate them for each other. If successful, the attacker is able to learn their communications and alter them. These attacks are prominent in the cases where the communication entities, for example a server and a client, have to agree on an encryption key for the first time without the knowledge of either of their public keys. This is because the client presents its public key to the server which may likely be new to it and hence could be easily replaced by an attacker. The client when presented with the server's key may not know if it is indeed the server's key and is not replaced by the attacker.

Modern secure web communication protocols such as Transport Layer Security (TLS) used in the HTTPS protocol depend upon Public-Key Infrastructure (PKI) to defend against these attacks. PKI enables communication entities to obtain digital certificates from trusted third parties, referred to as Certification Authorities (CA) as an endorsement of their identity. Through a digital certificate, the CA cryptographically conveys that the certified party has a particular name, public key and communication address. Since a digital certificate could be obtained from anyone, the CA should be trusted by both the communication entities for their certificates to be valid. With digital certificates in place, the attacker in the middle has to impersonate either the CA or impersonate the public key and the communication address, either of which are theoretically hard.

In practice, however, a communication entity trusts many CAs. For example a modern web browser such as Firefox or Internet Explorer trusts up to 100 of CAs. This is because of political and commercial reasons behind the PKI: each country or a big organization wants to have its own CA. Since the PKI does not limit the validity of certificates issued by a CA locally to some region or domain, a CA which is trusted by a communication entity can issue valid certificates to anyone. This means that 1) if an attacker succeeds in placing himself as a trusted CA list or 2) if he attacks and compromises a certification authority, he can again perform the man-in-the-middle attacks. Such attacks on PKI are not rare [HRE+14].

2.4 Route hijacking

Information travels on the Internet as packets that follow routes from machine to machine. The Internet itself 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) [RLH06].

The Border Gateway Protocol does not ensure that BGP routers use the AS number that has been allocated, or that the AS holds the prefixes it originates. So a router can be configured to advertise a prefix from an address space belonging to another AS in an action known as *IP prefix hijacking* [BFM+10]. This action can happen in the forms:

- Hijack the entire prefix – where 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. For example if the victim is announcing 200.200.0.0/16 the attacker announces 200.200.200.0/24. Due to the longest prefix matching rule, an AS that receives these announcements may direct traffic toward the wrong AS. This type of hijacking is associated to blackhole attacks, where the malicious AS drops all the packets received.

- Interception hijack – the attacker announces a fake route to an AS. The AS now forwards traffic from a victim intended to a legitimate destination to the malicious interceptor. The contents of the intercepted traffic can be analyzed/changed, before being sent to the legitimate destination. Figure 1 illustrates this type of attack.

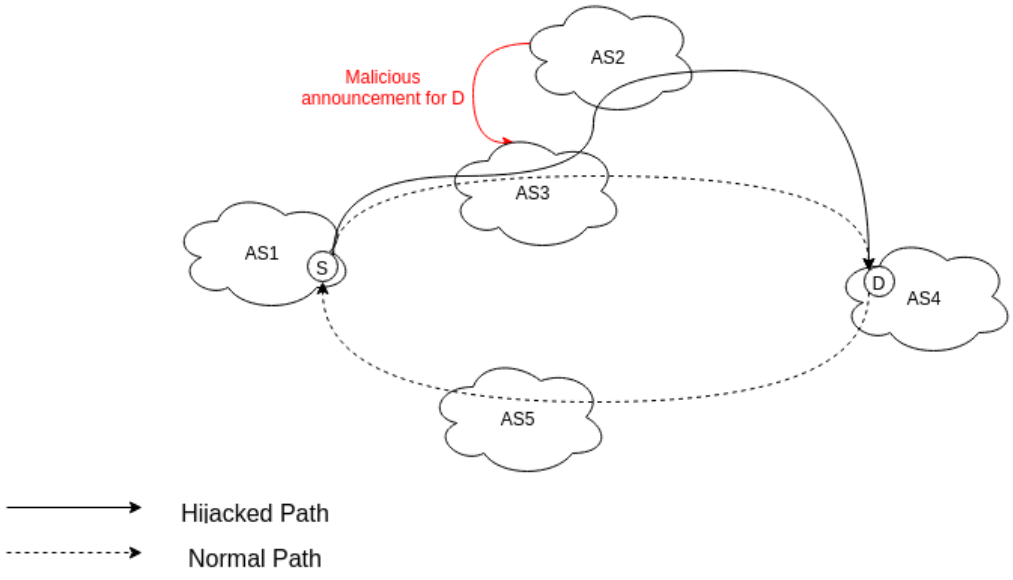

Figure 1. AS route-interception attack.

Schlamp et al. described an attack where an offender claims ownership of an entire AS [SCB13]. To perform an AS hijacking attack, the attacker pretends that he owns the AS of the victim. These types of attacks are harder to detect because unlike the prefix hijacking attack, there are no signs of duplicate origin announcements, the only change

that does occur is the formation of a new link to the upstream provider from the victim AS. According to the authors, to perform this attack, the offender needs to have a router configured with BGP and prove the ownership of the victim AS, to an upstream provider, by controlling Regional Internet Registrars (RIR) databases where the information about ownerships is stored. The authors conclude by suggesting an early detection system that combines multiple data sources and verifies the expiration date of the domain of the autonomous systems, sending a warning to ASes in which an expiry date is close. This is important because, if a domain expires, an attacker can re-register that domain claiming the ownership.

Autonomous systems are bound by business relationships, therefore network operators specify routing policies that affect which BGP routes are chosen. BGP UPDATE messages contain route attributes that are used by BGP routers to compare the announcements received. Some of the most important route attributes are the local preference, the AS path length and the origin type. A BGP router selects a route with a maximum value of local preference and a minimum value for the AS path length.

A survey [GSG13] was conducted in order to obtain some information about the routing policies in place, in which almost 100 responses from network operators were obtained. The questions asked involved mainly the usability of models of routing policies, like the Gao and Rexford model, and the criteria of BGP decision process (steps which help decide the route to choose). In the Gao and Rexford model, ASes that buy transit services, to obtain access to other parts of the Internet, are called customers, ASes that provide these services are named as providers, and finally ASes at the same level are known as peers. The model assumes the following conditions:

- By having a choice, the ASes always choose to route traffic to neighboring customers instead of a neighboring peer, or provider. This preference is due to the monetary gain obtained by choosing customer routes and the way to explicitly define it is by using a criteria known as local preference.

- ASes only export providers or peers routes to neighboring customers. This implies that an AS only exports traffic if it was paid to do so.

According to the responses, 68% applied both conditions and 19%only applied the first condition. Reasons registered for the non-usability of the export condition include secret agreements and that export restraining techniques may end up breaking routing. These evidences show that it is difficult to predict the paths that packets take due to the heterogeneity of routing policies in different ASes.

Autonomous systems can modify forwarding attributes for their own convenience. For example, they can:

- Reduce an AS path in order to look more attractive according to some routing metric;
- Put additional AS hops at the end to make a hijacked route look like it was originated by the proper AS;
- Add the victim AS to the AS path, and once the advertisement reaches the victim AS the BGP looping system will drop the misleading announcement [BFM+10].

BGP security today mainly consists of filtering suspicious BGP announcements, like 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. Therefore, having a global view of correct routing information would make it much easier to detect invalid routes.

An accurate routing registry would have prefix ownership, AS-level connectivity and routing policies enabled in each AS, helping ASes in verifying 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. Moreover the registry itself is often untrusted due to its power to manipulate the route information at will.

Ultimately, the adoption of security solutions is limited by the lack of sharing of reliable information in public Internet registries of about the correct mapping of IP addresses to ASes.

3 Services

The SafeCloud middleware will provide a set of four *services* that solve or mitigate the threats presented in the previous section. The services correspond to Tasks T1.2 to T1.5 of WP1, as shown in Table 1. Task 1.1 defines the middleware architecture and its first outcome is the present deliverable, D1.1. The table also summarizes the threats handled by each service.

Service	Task	Section	Threat handled
Vulnerability-tolerant channels	T1.2	3.1	Secure channel component vulnerabilities
Protected service provisioning	T1.3	3.2	Service identification
Route monitoring	T1.4	3.3	Man-in-the-middle attacks, route hijacking
Multi-path communication	T1.5	3.4	Man-in-the-middle attacks, route hijacking

Table 1: Correspondence between the middleware services, WP1's tasks, and the threats in Section 2.

Figure 2 presents these services in the context of the SafeCloud framework. In particular, the *secure communication* (WP1) is described in the second row and contains three *solutions:*

(1) Vulnerability-tolerant channels – this solution corresponds to the first service, which has the same name (T1.2).
(2) Protected services – this solution includes solution 1, and it extends it with the second service, protected service provisioning (T1.3).
(3) Route-aware channels – includes solution 2, and extends it with route monitoring (T1.4) and multi-path communication (T1.5).

	State of the Art	SafeCloud framework		
Secure communication	TLS secure channels	Solution 1: Vulnerability-tolerant channels	Solution 2: Protected channels	Solution 3: Route-aware channels
		Gives: tolerance to vulnerabilities in components (e.g., in hash function)	Gives: improved confidentiality thanks to a lesser chance of fake certificates; resistance to port scans and enumeration of network infrastructure (notary and portknocking)	Gives: improved confidentiality with warnings about route hijacking and making harder access to communication
		API: extended secure socket API	API: extended secure socket API	API: extended socket API
		Technology provided by: INESC-ID/TUM	Technology provided by: INESC-ID/TUM	Technology provided by: INESC-ID/TUM
Secure storage	Encrypted database	Solution 1: Distributed encrypted filesystem	Solution 2: Long-term distributed encrypted document storage	Solution 3: Secure block storage
		Gives: file storage	Gives: long-term (entangled) document storage (immutable)	Gives: block storage on individual data centers
		API: POSIX	API: REST (S3 or similar)	API: Key/value
		Technology provided by: UniNE / INESC-ID	Technology provided by: UniNE / INESC TEC	Technology provided by: UniNE / INESC TEC
Secure queries	None	Solution 1: SQL on encrypted values	Solution 2: SQL on encrypted keys and values	Solution 3: SQL on encrypted keys and values with untrusted clients
		Gives: privacy of data values against server	Gives: privacy of data and keys against server	Gives: privacy of data and keys against client and server
		API: SQL	API: SQL	API: SQL
		Technology provided by: INESC-TEC	Technology provided by: INESC-TEC and CYBER	Technology provided by: CYBER

Figure 2. The SafeCloud framework.

The following sections present the four services. Each section starts with a description of the mechanism, then presents the main components that implement the service, finally presents the service API.

3.1 Vulnerability-tolerant channels

3.1.1 Description

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

Several protocols to implement secure communication channels exist nowadays, such as TLS, IPsec or SSH. Each of these channels is used for a different purpose, but with the same goal of securing the communication. We introduced TLS in Section 2. The 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]. The Secure Shell (SSH) is an application layer protocol, similarly to SSL/TLS, that is used for secure remote login, secure file copy, and other secure network operations over an insecure network [YL06].

A secure communication channel becomes insecure if a vulnerability is discovered in its specification or implementation. Vulnerabilities may concern the protocol's specification, cryptographic mechanisms used by the protocol or specific

implementations of the protocol. Many vulnerabilities have been discovered in SSL/TLS originating new versions of the protocol with renewed security aspects such as deprecating cryptographic mechanisms or enforcing security measures. Certain implementations of SSL/TLS have also been considered vulnerable by having implementation details causing a breach in security and affecting devices worldwide.

Our solution is a secure communication channel tolerant to vulnerabilities which does not rely on only one cipher suite. It is our belief that, in order to solve the problem, *diversity* and *redundancy* are helpful to mitigate vulnerabilities. Diversity and redundancy consist in using two or more different mechanisms with the same objective. For example, MD5 and SHA-3 are both hash functions used to generate digests. In a real case, where MD5 has become insecure, our secure communication channel resorts to other mechanisms, such as SHA-3, in order to keep the communication secure. Using diversity and redundancy, when a mechanism is targeted by an attack, another mechanism is able to maintain the security and availability of the communication.

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

One of the challenges of this work is to evaluate if diversity and redundancy have a real impact on increasing the security of a communication channel while still having reasonable performance.

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

Our proposal solves the problems originated from having only one cipher suite negotiated between client and server. In the case when one of the cipher suites mechanisms is insecure, the secure communication channels using that cipher suite may be vulnerable. Our diverse secure communication channel negotiates more than one cipher suite between client and server and, consequently, more than one encryption mechanism will be used for each purpose. Hence, the channel does not rely in only one cipher suite. Although most cipher suites used by TLS 1.2 are regarded as secure, there is no assurance that an agency or company with high computational power is not able to break that encryption mechanism today or in the near future.

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

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

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

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

3.1.2 Components

The channel is implemented by two components: the server and the client. Just like in TCP, the server awaits connection requests, and the client takes the initiative of contacting the server for establishing a connection. Just like in TLS, there is an initial handshake protocol between client and server that is intended to exchange information about the identities and supported cryptographic protocols, to authenticate the communicating parties and to perform the distribution of session keys.

3.1.3 API

The classical programming interface for TCP and UDP is the Berkeley UNIX Socket API [S90]. It provides primitives like *socket* (to create a communication endpoint), *bind* (to associate an IP address and a port to a socket), *send* and *sendto* (to send data), *receive* and *receivefrom* (to receive data) and *select* (to block waiting for receiving from several endpoints).

The API for Secure Sockets (a socket that uses SSL/TLS with protection guarantees) is provided in the Java programming language in the javax.net.ssl package. The classes extend regular sockets and add a layer of security over the underlying network transport protocol.

The typical server code consists in:

- SSLServerSocketFactory.getDefault()
- sslServerSocketFactory.createServerSocket(portNumber)
- sslServerSocket.accept()
- sslSocket.getInputStream()

The typical client code consists in:

- SSLSocketFactory.getDefault()
- sslSocketFactory.createSocket(hostName, portNumber)
- sslSocket.getOutputStream()

The creation of a secure socket is done through a factory object that abstracts the concrete class that is instantiated. This indirection level can be leveraged to return

modified secure sockets, like our proposed vulnerability-tolerant secure sockets, while maintaining overall API compatibility.

A secure server socket is created by providing the desired binding with usually the port number. The accept method leaves the server awaiting connection requests.

Reading from and writing to the socket is performed using the Input and Output stream objects that are found in the java.io package.

When secure sockets objects are first created, no handshaking is done so that applications may first set their communication preferences e.g. what cipher suites to use. However, security is always provided by the time that application data is sent over the connection because the security handshake is done implicitly once the client tries to read from or write to the socket.

The handshake can be also be explicitly started by calling the startHandshake() method. This method can be extended to add options related with vulnerability-tolerance, if the default values need to be modified. The managing of the cipher suites will also need to be modified, as the supported and enabled cipher suites will have to deal with multiple simultaneous choices.

3.2 Protected service provisioning

3.2.1 Description

Protected service provisioning addresses the threat that attackers can identify the services running on a host to determine their attack vector. This is done by employing port-knocking tactics.

Port-knocking limits the availability and visibility of a service only to authorized clients. An advantage when compared to other authentication mechanisms is that with port-knocking unauthorized clients will neither be able to access the service nor be able to determine if the service is even available and running on the host. This is because port knocking works at the transport layer in OSI networking stack. In the case of a TCP connection attempt to the protected service, the first TCP SYN packet the TCP handshake contains a unique authentifier authenticating the TCP connection. Whereas in case of a UDP-based connection, the UDP header could contain the authentifier.

Since neither TCP nor UDP protocols incorporate fields for authentifier data, it is accommodated in fields which allow arbitrary values, for example, the initial sequence number field in TCP. In the case of UDP only the source and destination port number fields can have arbitrary numbers, which restricts the number of bits for authentifier data.

The following factors are considered while devising our port-knocking mechanism:

- **Protection coverage**: should only some subset of services running on the host be protected or all of them? This question impacts the design decisions to provide port-knocking authentication of each service or to just one proxy service running on the host.
- **Authentication mechanisms**: How is the authentication between hosts achieved? Is this done through a PKI based on certificates where the cloud

service provider deploys a CA and certifies all of the hosts and the service users? Or is it done through sharing passphrases to individual service users and also among hosts? This impacts whether we could reuse off-the-shelf port-knocking solutions such as knockknockd, which currently only supports passphrase based authentication.

- **Deployability**: how far are service providers and users willing to go to configure the operation systems on their hosts? The answers to this question lets us rule out solutions involving complex configurations which involve patching the operation system's kernel.
- **Scalability**: there are off-the-shelf software solutions providing port-knocking, knockknockd is an example of this. However, it requires each client to be authorized by a passphrase stored in a file. While this approach is simple, this limits its scalability as the knockknockd service has to scan through the passphrases each time a port is knocked. Also, it hinders deployment scenarios where a service provider has multiple servers, for either load balancing or redundancy, and a client should be able to access any of those services with its port-knocking passphrase. In this case, the passphrase profiles of clients have to be synchronized on all servers providing access to the clients. Also, when a client has to be permitted or restricted, its corresponding passphrase profile has to be added or removed respectively.

Considering these factors, our solution is to provide a complete protection coverage extending to all system services, if need be. To address the scalability concerns, we propose to use PKI based authentication approach as this removes the requirement to synchronized passphrase profiles and only requires CA certificate to authenticate any client. The requirement of synchronizing passphrase profiles is in this case replaced by having a requirement of implementing revocation lists. However, there exist frameworks for dealing with revocation lists and provided that we choose a reasonable certificate expiry dates, we can bound the size of the revocation lists.

3.2.2 Components

3.2.2.1 Protocol

The protocol for port-knocking involves a one-way handshake between client and server. The handshake is always initiated by the client. As part of the handshake the client transfers its certificate in the payload of an UDP packet to a pre-determined port on the server. This payload will also contain the port number the client wants to open on the server, current timestamp, IP address and port number of the client's transport endpoint, and a HMAC over these fields. The payload is encrypted with the server's public key. If the port knocking service on the server determines the certificate to be valid, the client is said to have successfully port-knocked.

If the UDP packet is malformed or the client's certificate is invalid, the server immediately responds with an ICMP port unreachable message to the client, giving the impression that the UDP port is closed. For an attacker or an unauthorized client, this message conveys that no service is using the given port.

In the case of a successful port-knock by a client, the server does not respond back to the client's first UDP packet. It instead waits for the client to send its connection packet to the intended port. Any packets to this port received from the client within a period of time will then be allowed by the firewall. The port is closed for the client after a timeout

for TCP and UDP connections or by explicit connection teardown in the case of TCP connections.

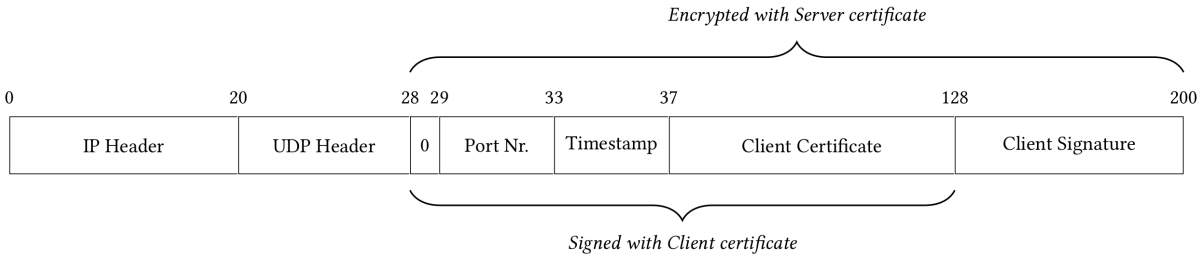

Figure 3: Format of the UDP knock packet

The format of the UDP knock packet is shown in Figure 3. The format is not finalized at the time of writing this deliverable. Since the format impacts the implementation which is however abstracted by the client library API, any changes in the payload format are less likely to impact higher layers which use the provided API. Version incompatibility for the payload format is addressed by being as backwards compatible as possible.

A handshake packet from a client is considered malformed if:

- It is not a UDP packet;
- It is smaller than a minimum length required by the protocol;
- The destination port is not the pre-defined port on which the port-knocking service will be expecting to receive it;
- The payload fails to decrypt with the server's private key;
- The decrypted payload does not match the payload format;
- The decrypted payload has a different version than those supported by the server;
- The client certificate is in unknown format.

Additionally, the certificate in the decrypted payload is treated as invalid if:

- The client certificate has expired or the validity starts in future;
- The client certificate is not signed by a CA trusted by the server;
- The client certificate is in a latest CRL issued by the server.

3.2.2.2 Server component

This is a service running on the server providing port knocking for the services running on the same server. It interacts with the firewall of the system to determine which clients are to be authorized. To participate in the port knocking handshake, the firewall is made to log all incoming UDP packets to a pre-determined port. The service will read these packets and discard malformed ones by responding with ICMP port unreachable messages to their respective senders. These messages are also sent if the packet is well-formed but the client certificate is invalid.

For well-formed packets containing valid client certificates, the service opens up the intended port on the firewall for the respective client.

The server component is configurable via a configuration file. The configuration includes the port number of the UDP handshake packets, a list of CAs whose certificates should

be a treated as valid, and a certificate revocation server address for periodically downloading the certificate revocation lists.

3.2.2.3 Client library

Client applications have to implement the port-knocking protocol before they can access the services on a server implementing our port-knocking solution. The client library implements this client-side protocol of the port-knocking protocol. This helps in code reuse and maintenance as any minor changes in the protocol only concern with changes to the library instead of the client applications.

The programming interface of this library is described in the API section below.

3.2.2.4 Proxy

The proxy is a service running on the client side. It is used solely for convenience and is implemented to improve deployability. It provides transparent access to services on a port-knocked server without requiring the client application to use the client library.

This works by having the client application connect to the proxy's local SOCKS port and asking the proxy to open connections to a port-knocked service. The proxy then does the port knocking handshake by using the client library and forwards the application traffic the between the client application and the port-knocked service. By using the proxy, the client application need not require to interface with the client library to access port-knocked services.

3.2.3 API

The API for the client library will provide a single function call, which inherits from the BSD socket API's open function call.

```
int knock_open(AF, TYPE,
               client_cert, server_cert,
               server_addr, knock_port, service_port)
```

1. AF: Communication address family. Used to select either IPv4 or IPv6 sockets. Only AF_INET and AF_INET6 are supported
2. TYPE: STREAM or DGRAM for TCP or UDP respectively
3. client_cert: client's certificate to use in the port-knocking handshake
4. server_cert: server's certificate
5. server_addr: socket address for server. This should correspond to the AF parameter.
6. knock_port: the port to which the initial UDP packet has to be sent to
7. service_port: the port of the service to which the connection has to be granted after successful port-knocking at the server
8. Return: -1 upon error; fd >0 for a file descriptor corresponding to the socket

This function tries to port knock the server identified by server_cert and is reachable on server_addr. It does so by first sending the first UDP packet of the handshake to knock_port. If the requested TYPE is STREAM, then it also connects the socket to the service_port on the server.

The return value of the function is a file descriptor corresponding to the underlying socket. In the case of TCP, i.e., when TYPE is STREAM, the socket will be in a connected state.

Caveats: the connection to the port-knocked service may expire after a period of inactivity. This will be more relevant if UDP (TYPE set to DGRAM) is used, for TCP implements TCP keep-alives as part of its stack.

3.3 Route monitoring

Our approach to route monitoring involves various techniques to collectively deduce any anomalies in the routes used by SafeCloud endpoints. For this we use network measurements deriving from actively probing the characteristics of the routes and, optionally, passively inferring routing changes from BGP route advertisements as described by Feamster et al. [FAB+03]. The collected data is then analyzed on the endpoints, and a response is triggered by routing anomalies.

Through active probing we determine the number of involved hops, latency and packet loss in the route. This requires traffic to be exchanged on the route periodically. Usually, this traffic is generated by the application layers, however, in the case the upper layers do not supply enough traffic, active probing requires dummy traffic to be sent to keep the route characteristics updated. The measurements determined in this way correspond to the amount of traffic in the route. With a varying traffic rate, the measured route characteristics vary due to queuing and scheduling involved at the route's hops.

This work does not intend to substitute the use of best practices to configure BGP, or several prevention mechanisms that have already been proposed [BFM+10] [KFR06], but to present a route hijacking detection system using a set of monitoring techniques like, traceroute, measuring latencies and IP traceback mechanisms that can effectively and reliably monitor the routes the packets are taking, ultimately leading to a conclusion about the existence of a route hijack.

Moreover, since routing in the Internet is susceptible to changes due to varying network load and connectivity conditions, not all routing anomalies are to be treated as attacks on the routing protocol. This causes difficulty for detecting attacks automatically, while keeping the false positive rate low is aggravated by the fact that we measure the route characteristics from the endpoints. Hence, our route monitoring approach does not attempt to identify such attacks, however it will provide mechanisms to statistically evaluate the route measurements and trigger response callbacks from applications upon finding an anomaly. It is left up to the applications to either log the anomaly for future investigations or take preventive actions anticipating a possible routing attack.

3.3.1 Description

The objective of this service is to provide a route monitoring system. The system employs various techniques to measure route characteristics in a comprehensive way such that they are harder to be manipulated by an active attacker in the route. Additionally, we consider the following requirements:

- *Efficient* – The overhead associated to the measurements has to be kept as low as possible;
- *Scalable* – The monitoring system has to scale well i.e., the number of routes being monitored shall not become a bottleneck for the performance of the system
- *Easy to deploy* – The system shall not depend on vantage points and privileged information like BGP messages that limit its ability to be used in practice;
- *Resilient* – The techniques that constitute the system have to be redundant enough to prevent attackers from manipulating the route characteristics;

Our monitoring system takes into considering three important route metrics: latency, hop count, and the path between the source and destination.

Since packets in a route take non-negligible amount of time to traverse a route, an attack causing the packets to traverse a longer route would increase the delay. However due to the congestion in the network, caused by legitimate reasons, this method of detecting route hijacking is not reliable. For this reason, we propose to also consider the number of hops in the route along with the delay. The number of hops in the route could be measured reliably by observing the Time-to-Live (TTL) field of the Internet Protocol (IP) packets at the source and destination. An attacker that has hijacked the route can bypass this monitoring technique by adjusting the TTL field of the IP packets in order to reproduce the number of hops of a normal path between the source and the destination. Therefore, our proposal will keep state both of the number of hops and the time that packets take from end-to-end.

An anomaly caused by the two states, latency and hop count, gives us enough reason to start investigating the path the packets are taking. Furthermore, less extreme change cases, like 20-30% changes in the metrics, may also require investigation. For this, traceroute is used but routers may be configured to block traceroute requests or certain protocols like ICMP or UDP. Thus traceroutes with different protocols are executed and the results are merged generating the most complete path possible. Any difference in the observed path further confirms the anomaly.

3.3.2 Components

The route monitoring system comprises of a monitoring service run on top of SafeCloud communication endpoints. The service comprises of the following components:

- TTL monitor
- Route metrics monitor
- BGP monitor (optional)
- Anomaly detector

The TTL monitor estimates the number of hops involved in the route. For this task, it uses a multitude of approaches based on Traceroute (ICMP, TCP, UDP), Secure Traceroute [PS03], IP options (Record Route, Strict Source Record Route, Loose Source

Route), and estimations based on Return TTL. None of these approaches are guaranteed to work universally. This is due to the fact that each hop in the involved route can be configured independently and behaves differently for each approach. For example, some IP routers are known to discard IP packets with any of the Record Route options by default; some do not respond to ICMP traceroute requests, but will respond to UDP traceroute. Therefore, each of these approaches will be implemented as a pluggable module. This helps us to prioritize implementation of some of them and to develop and to test them individually. It also helps us to compare results of each module and thus determine the hop count with good confidence should multiple modules determine the same number of hops.

Route metrics monitor is tasked with the responsibility of determining the latency, bandwidth and packet loss in a network. The metrics are observed for each or some of the traffic sent by higher layer applications. In the case where the metrics needs to be kept updated without application layer, dummy traffic is used. The observed metrics for the individual packets are subjected to statistical filtering with bounded history to even out noise.

BGP monitor observes BGP updates for anomalies and anticipated changes in existing routes. BGP updates are collected from different vantage points such as PlanetLab [C+03] and/or 3rd party providers such as Route Views Project[1]. Since BGP data is not expected to be available in most deployment scenarios, we consider it as optional.

The anomaly detector maintains a history of inputs from the monitors and processes them through statistical analysis. This analysis considers the available data, both current and historical, to identify changes to the route. If a change is perceived, an alert is propagated to the application using the API.

3.3.3 API

The API consists of calls to get information about metrics of a route. Since these are statistical measurements, the calls return the metrics and predicted error bounds. This information is provided synchronously. The API also supports registering callbacks so that applications can be notified asynchronously through registered callbacks upon specific events.

```
route_register(source, destination, modules)
```

This function registers a route between given source and destination to be monitored. It instantiates the route object with the necessary monitoring modules.

1. `source`: the source address
2. `destination`: the destination of the route
3. `modules`: list of various route monitoring modules to be used for monitoring this route. The choice determines the route monitoring metrics that will be available for this route.

```
route_get_info(route, metric_type)
```

1. `route`: the route object which is created from an earlier call to route_register

[1] https://www.routeviews.org/

2. `metric_type`: the type of the metric to be used. This depends on the modules registered for monitoring the route in the call to route_register
3. `return`: the corresponding route metric object. If no metric type is available, an error or an exception is thrown.

```
route_monitor_callback(closure, event, route_metrics)
```

This is the type of callback to be implemented by an application interested in monitoring the route. After implementing this callback, it should be registered with a call to `route_monitor()`. It will then be called asynchronously whenever an interesting event happens. The events could be a change in the number of hops, change in the TTL number, etc.

1. `closure`: a pointer to a block of memory which is registered along with this callback. This is used to associate the callback with some state information defined by the application.
2. `event`: the event due to which this callback is called
3. `route_metrics`: the associated metric object.
4. `return`: nothing.

```
route_monitor(route, events, monitor_callback, closure)
```

Register a monitoring callback for a route.

1. `route`: the route object for which this monitoring callback has to be registered.
2. `events`: set of events of interest for the caller. An event from this set will trigger the callback
3. `monitor_callback`: the callback to be called when an event happens
4. `closure`: State which should be associated with the callback. It will be available in the callback as the first argument when the callback is called.
5. `return`: a monitoring handle which can be used to cancel the monitoring of this route and thereby unregistering the callback.

```
route_monitor_cancel(monitor)
```

Cancel a previously registered monitoring callback

1. `monitor`: monitoring handle previously obtained from route_monitor.

3.4 Multi-path communication

3.4.1 Description

Multi-path communication consists in dividing the data to be sent over a network and split it across two or more paths, spreading on the source and sinking on the destination. This idea may allow providing data confidentiality even if the attacker can break cryptographic protocols like TLS.

A problem faced by implementing multi-path routing is the possibility of having a limited number of channels where to distribute the information due to path failures. For example, assume a network composed by ten possible paths from a certain source to destination, where only two paths are available to send the data. These path failures can be caused by different reasons, but, in this case, it is important to study the denial of service or selective forward as the main malicious cause for these failures [SP10].

This service provides an additional layer of security by distributing the communication through several channels. In order to do so, four mechanisms can be leveraged. The communication starts with *path discovery*, where network nodes (routers or other devices with routing capabilities) are discovered, i.e., the source acquires information on which nodes it can send the information through. This discovery will be based on an upper and lower bound on geographic location. Using a topology-aware and trust-based decision algorithm, several nodes from different Internet Service Providers are chosen, according to their location and trust level – this approach is called (1) *Multi-homing*. The trust level is calculated according to the amount of packets dropped and response time to the source in the discovery phase. This selection and the number of nodes used will be based in the amount of physical paths available and the amount of information to be sent. Each node will create a single-hop overlay link to the destination, generating an (2) *Overlay Network*. Successively the packets are split among the different overlay links that were previously created and sent to the destination using Multi-path TCP (3) *MPTCP* [FRHB13]. This protocol is an extension of the generic TCP implementation that has the ability to distribute and send data between different interfaces or IP-addresses. Therefore, it will be used to distribute the data among all links in the overlay network. In the case when not enough channels are acquired to provide resilience against the interception of packets, (4) *Network Coding* is applied to the packets in order to provide some resistance to these eavesdropping attacks.

Some requirements for the service are:

- The path selection should be as physically disjoint as possible;
- The path selection should be randomized according to lower and upper bounds of geographic location;
- The framework should resort to network coding when there are not enough paths to use;
- The security of communication should always overcome the performance, however the performance should be acceptable.

3.4.2 Components

Implementing this architecture requires a set of components other than the communicating endpoints.

For implementing application-layer routing a set of nodes have to be used to route the data. We call these nodes *relays.* These relays will typically be servers in the clouds that implement the SafeCloud architecture. Nevertheless, additional computers may be obtained for that purpose. In this case, it is useful to increase the diversity of the available channels. These computers can be rented to additional cloud service providers, or provided by interested users (e.g., large companies committed to supporting the SafeCloud architecture).

The second component is a *membership service* that tracks which nodes are available to play the role of relays at each moment. This component is critical for the availability of the communication, so it has to be secure, dependable, and disaster-tolerant. We envisage to implement this service on top of a coordination service called DepSpace [BAC+08]. DepSpace is replicated so if these replicas are scattered geographically it can continue operating, even if disasters occur. Moreover it uses Byzantine fault-tolerant protocols to provide data integrity and availability despite intrusions in a subset of the

servers. This service will keep a list of active relays, removing them from the list if they are unavailable. DepSpace is a tuple space, not a membership service, but it can be programmed for that purpose.

3.4.3 API

The API of the service is again essentially the sockets API, with two extra functions to define and get the number of channels to use:

- setNChannels()
- getNChannels()

4 Architecture

This section presents the first version of the *private communication middleware architecture*, the core of this deliverable. This architecture expresses how the services will be deployed, so it may still be revised depending on the evolution of the project. The final version of the architecture will appear in deliverable D1.3.

The purpose of the SafeCloud middleware is to provide private and secure unicast communication within the SafeCloud architecture. In terms of privacy/security the middleware aims to provide four properties despite the threats explained in Section 2:

- *Confidentiality or Privacy*[2]: absence of disclosure of data to unauthorized parties;
- *Integrity*: absence of unauthorized message modifications;
- *Authenticity*: absence of personification of message senders (i.e., the message sender is the one indicated in the message);
- *Availability:* the middleware shall be ready to provide its service when requested.

4.1 Middleware entities

From Section 3 it is possible to extract the main entities of the architecture:

Endpoints. These are the computers involved in the communication: servers, desktops, laptops, smartphones, tablets, etc. The communication takes two basic flavors: *machine-to-cloud* (i.e., between external endpoints and cloud endpoints), and *cloud-to-cloud* (i.e., between endpoints in clouds).

Public key infrastructure (PKI). The middleware assumes that every endpoint has a long-term public-private key pair, with the key size selected to be secure in the future (i.e., "expected to remain secure in 10-50 year lifetime") according to ENISA's recommendations [Eni14]. The public key of this key pair is distributed using certificates through a PKI. The PKI is composed of several components [CSF+08]:[3]

- *Certification authorities* (CAs): entities that generate certificates; CAs should be replicated using intrusion-tolerant schemes [ZSR02,VCB+13] so they can be resilient before strong adversaries;
- *Registration authorities* (RAs): components to which CAs may (optionally) delegate certain management functions, e.g., the enrollment of users;
- *CRL issuers*: components that generate and sign certificate revocation lists (CRLs);
- *Repositories*: components that store and distribute certificates and CRLs.

Certificate notary (CN). This notary service will be provided by some SafeCloud deployments by running the Crossbear [HRK+12] notary service. The certificates of the

[2] The term privacy is also used in the sense of confidentiality of personal data. Here we consider it in a broader sense of confidentiality of any data being transported in messages.

[3] Endpoints (end entities in the RFC nomenclature) may also be considered part of the PKI but we leave them out for the purpose of considering the PKI a service.

notaries will be included with the middleware to avoid man-in-the-middle attacks against the notaries. The notaries provide the certificate verification service to SafeCloud endpoints as a defense against SSL man-in-the-middle attacks.

Relaying infrastructure. The multi-path communication service involves sending messages through different physical paths over the Internet. The IP protocol does not provide to endpoints the option to define the network layer path, so this has to be implemented at application layer, using an overlay network and multi-homing. The relaying infrastructure has two main components:

- *Relays*: the relays are the intermediate nodes of the overlay network, i.e., computers, typically servers, that support application layer routing for multi-path communication purposes. The relays will provide a simple forwarding service so, from the security point of view, they are critical mainly in terms of availability.
- *Relay membership service* (RMS): an online database of relays, which can be joined and left by relays that come online or go offline for some reason. From the security point of view this component is critical for availability so it is replicated using intrusion-tolerant schemes, similarly to CAs. This service can be implemented using an intrusion-tolerant coordination service like DepSpace [BAC+08].

4.2 Topological architecture

The topological architecture consists in deploying these components in the environment. Very roughly we can consider three basic realms:

- *SafeCloud clouds*: the SafeCloud clouds where data and processing are scattered for security purposes;
- *SafeCloud users*: users/consumers of the SafeCloud architecture, typically companies and other organizations or individuals;
- *Third-parties*: other organizations, e.g., a PKI or an organization contracted for monitoring or relaying; such organizations may even be cloud service providers, but their clouds are not SafeCloud clouds.

Figure 4 represents the abstract topological architecture of SafeCloud, with N SafeCloud clouds, N' SafeCloud user networks (organizations), and N'' third-parties. The Internet interconnects these networks. We say that this is an *application layer view* of the architecture in the sense that it only presents components that implement the whole OSI/Internet stack.

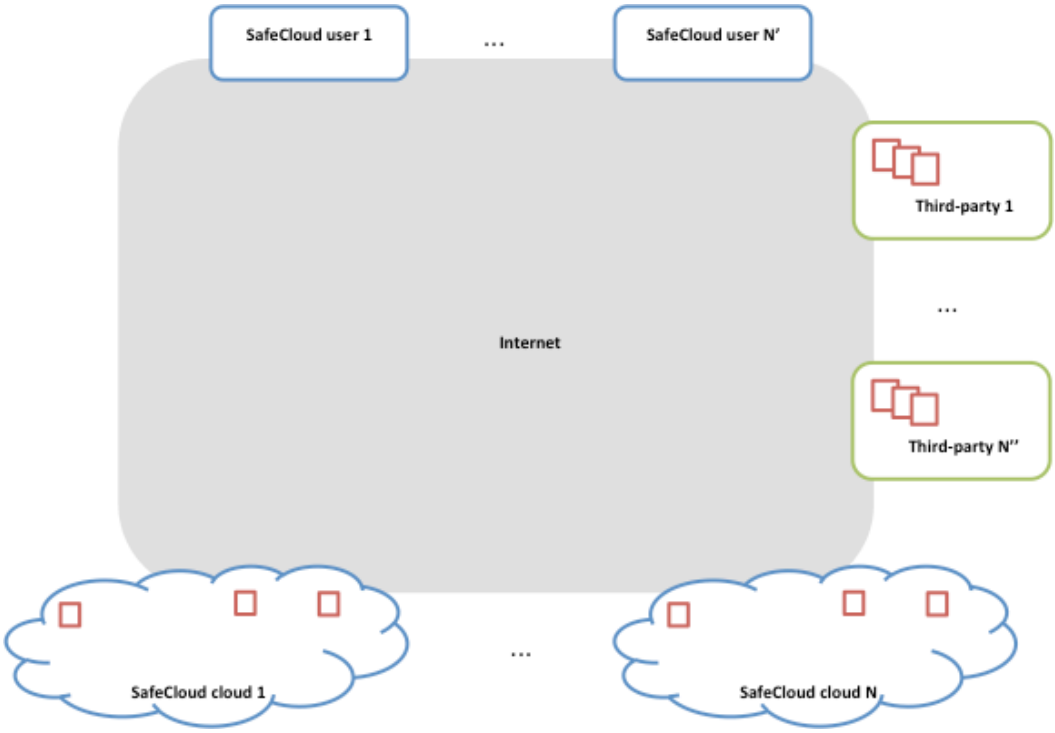

Figure 4. Abstract topological architecture: N SafeCloud clouds, N' SafeCloud user networks, and N'' third-parties, all interconnected by the Internet.

There are several options in terms of how the components of the previous section are placed in the realms:

- Endpoints: can be at SafeCloud clouds or users;
- PKI components: can be at SafeCloud clouds or third-parties;
- Certificate notary: services could be installed at some SafeCloud deployments or, alternatively run through third-party cloud service providers;
- Relays: may be at SafeCloud clouds, third-parties, or even at large SafeCloud users willing to commit to the operation of the SafeCloud architecture;
- Relay membership service: placed typically at SafeCloud clouds.

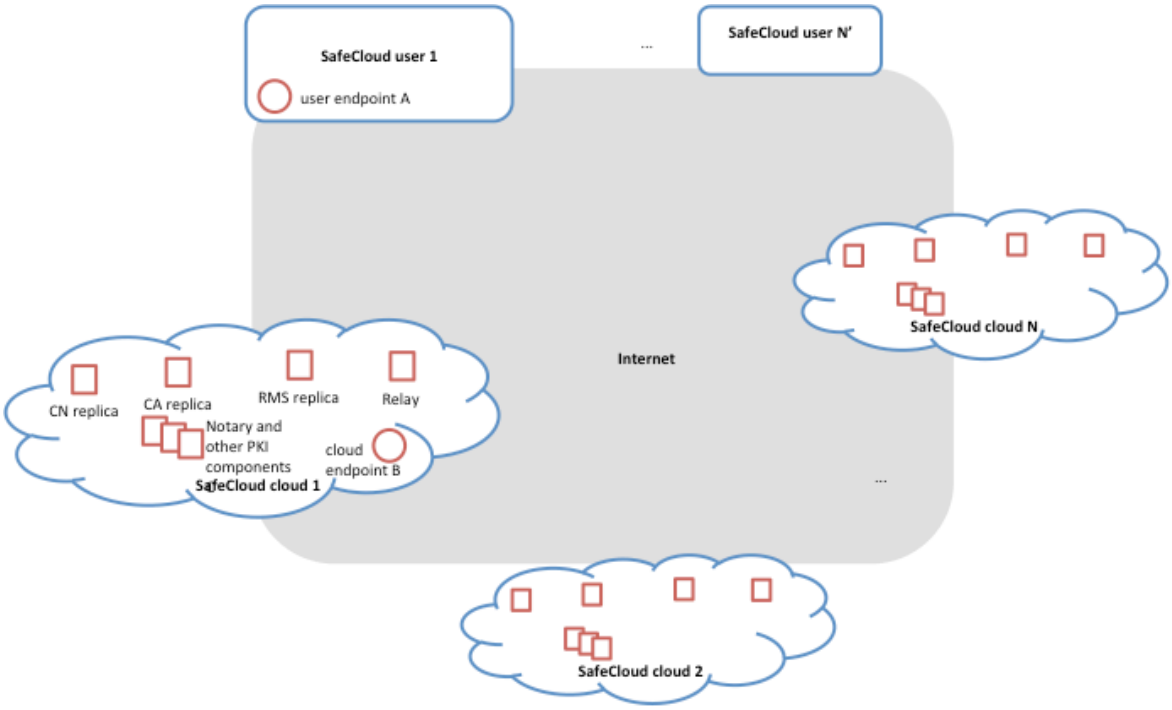

Figure 5. Topological architecture: application layer view; no third parties.

Figure 5 represents a more concrete topological architecture with N SafeCloud clouds and N' SafeCloud users, showing also the middleware services, some of them replicated, and a few endpoints. Figure 6 presents a similar architecture, now including one third-party that provides a PKI service.

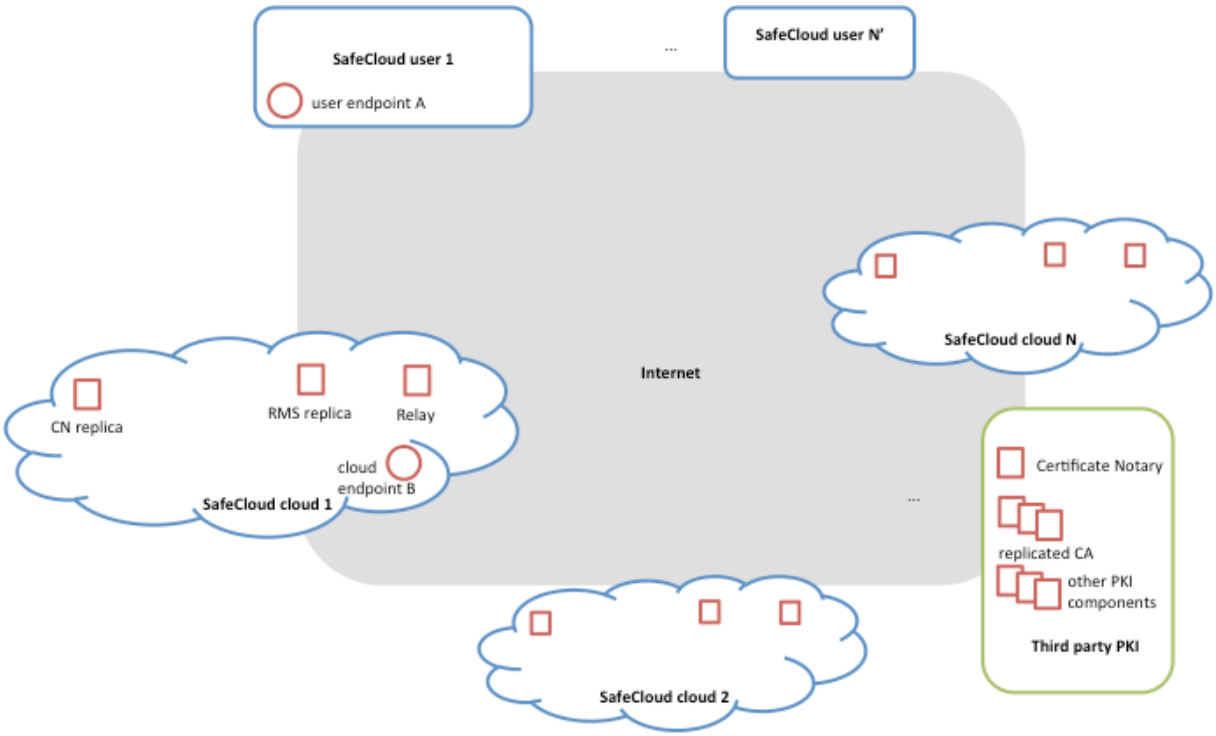

Figure 6. Topological architecture: application layer view; one third-party providing the PKI service.

Figure 7 provides a *network layer view* of the architecture in Figure 5, as it shows also devices that implement the protocol stack up to that layer. This means adding (network layer) routers to Figure 5. This allows us to show multi-path communication using different network paths, meaning different (network layer) routers and links.

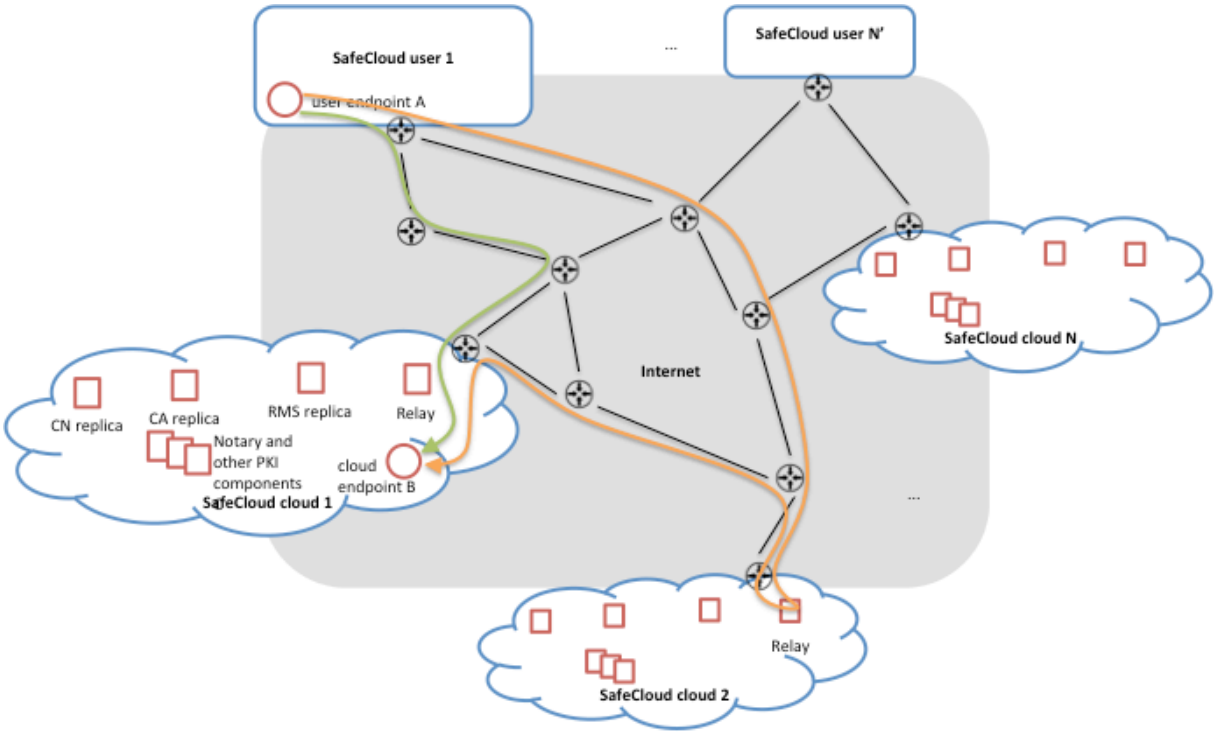

Figure 7. Topological architecture: network layer view; no third-parties. Endpoint A communicates with endpoint B using 2-path communication.

4.3 Software architecture

The software architecture will differ considerably between endpoints and other components such as RMS or CAs.

4.3.1 User endpoints

In endpoints the main SafeCloud software components are:

- SafeCloud Client Library (SCCL) – used to implement applications that use the middleware for communication; provides the API presented in Section 4.4.
- SafeCloud Client Agent (SCCA) – a process that runs in background and implements some of the services that require activity at the client side, e.g., monitoring the diversity and error rate of overlay paths.

Moreover we envisage the need of using the middleware to provide tunnels to transport the communication of legacy applications. The most obvious example is web communication. To support it we consider the following component:

- SafeCloud Web Proxy (SCWP) – it allows browsers to use SafeCloud's secure channels; it uses the APIs of the middleware, implemented by the SCCL.

Figure 8 represents these components.

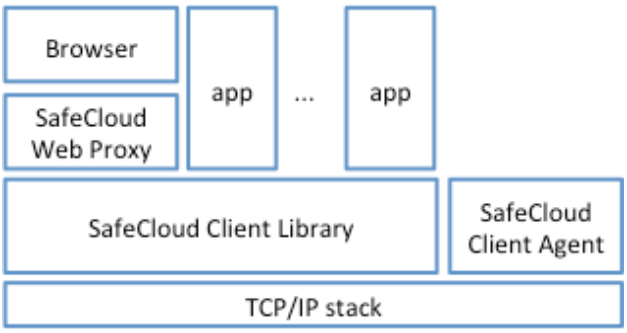

Figure 8. SafeCloud user endpoint software architecture with one browser and several applications using the middleware.

4.3.2 Other components

The non-endpoint components are implement as daemons running in servers. Several of these components can be instantiated in the same physical and/or virtual servers. For example, if the RMS and the CA both have 4 replicas, we can have only 4 physical servers, each one playing both the role of RMS replica and CA replica. However, security considerations have to be taken into account when sharing physical machines this way.

There is one daemon per component (e.g., one for relays, one for RMSs, one for CAs, etc.). However there are components that are used by all daemons:

- SafeCloud Server Library (SCSL) – server-side of SafeClouds' secure channels.
- SafeCloud Server Agent (SCSA) – a process that runs in background and implements some of the services that require activity at the server side, similarly to the SCCA.

There is also a specific daemon for supporting stock web applications:

- SafeCloud Reverse Web Proxy (SCRWP) – a server-side proxy that receives requests and sends replies using SafeCloud's middleware (secure channels) and calls web applications running in unmodified web or application servers (e.g., Apache, Tomcat, or WildFly/JBoss).

Figure 9 represents these components.

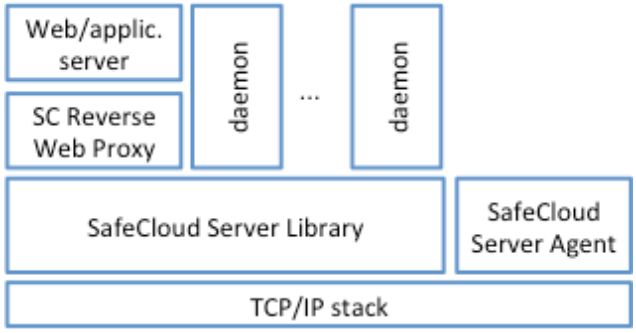

Figure 9. SafeCloud node software architecture with one web/application server and several daemons using the middleware (e.g., Relay, RMS, CA).

4.4 API

The API has two parts:

- Communication API (provides primitives to send and receive messages)
- Management API (provides primitives to manage the middleware and the communication, e.g., for monitoring, add/remove relays, etc.)

The functions of the API are those described in the previous section. Table 2 summarizes this information, mentioning the section that describes each part.

API Part	Service	API Functions	Section
Communication API	Vulnerability-tolerant channels + Multi-path communication	SSLServerSocketFactory.getDefault() sslServerSocketFactory.createServerSocket() sslServerSocket.accept() sslSocket.getInputStream() SSLSocketFactory.getDefault() sslSocketFactory.createSocket() sslSocket.getOutputStream()	3.1.3
Management API	Vulnerability-tolerant channels	–	3.1.3
	Protected service provisioning	knock_open()	3.2.3
	Route monitoring	register_route() route_get_info() route_monitor_callback() route_monitor()	3.3.3
	Multi-path communication	setNChannels() getNChannels()	3.4.3

Table 2: SafeCloud middleware API.

5 Conclusion

This document presents the first version of the SafeCloud middleware, which aims to provide the same properties as common *secure channels* (e.g., SSL/TLS, IPsec) – *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 (e.g., that a certain cryptographic algorithm is secure).

The deliverable presents:

- The **threats** that the middleware is aimed to handle: secure channel component vulnerabilities, service identification, man-in-the-middle attacks, and route hijacking;
- The **services** the middleware will provide: vulnerability-tolerant channels, protected service provisioning, route monitoring, and multi-path communication;
- The **architecture** of the middleware: its entities, the topological architecture, the software architecture at the nodes, and its API.

The architecture presented is necessarily preliminary, as the components are still being developed and there is still not a complete understanding on how they will work and interact. The final version will appear in deliverable D1.3.

6 References

[ABD+15] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS), Denver, CO, October 2015.

[BAC+08] A. Bessani, E. Alchieri, M. Correia and J. Fraga. DepSpace: A Byzantine Fault-Tolerant Coordination Service. In Proceedings of the European Conference on Computer Systems (EuroSys). April 2008.

[BBD+15] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, and M. Kohlweiss. A Messy State of the Union: Taming the Composite State Machines of TLS. IEEE Symposium on Security and Privacy, pages 1-19, 2015.

[BFM+10] K. Butler, T.R. Farley, P. McDaniel, and J. Rexford. A Survey of BGP Security Issues and Solutions. Proceedings of the IEEE, Vol. 98, N. 1, pp. 100-122, 2010.

[C+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. 2003. PlanetLab: an overlay testbed for broad-coverage services. SIGCOMM Comput. Commun. Rev. 33, 3 (July 2003), 3-12.

[CDF+14] M. Carvalho, J. DeMott, R. Ford, and D. Wheeler. Heartbleed 101. IEEE Security & Privacy, Vol. 12, N. 4, pp. 63-67, 2014.

[CSF+08] D. Cooper, S. Santesson, S. Farrel, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile (RFC 5280). May 2008.

[DH76] W. Diffie and M. Hellman. New Directions in Cryptography. IEEE Transactions on Information Theory, Vol. 22, N. 6, pp. 644-654, 1976.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol, Version 1.2 (RFC 5246), 2008.

[Eni14] ENISA. Algorithms, Key Size and Parameters Report – 2014. November, 2014

[F99] H. Feistel. Data Encryption Standard (DES). FIPS Pub 46-3, 3, 1999.

[FAB+03] N. Feamster, D. G. Andersen, H. Balakrishnan, M. F. Kaashoek. Measuring the Effects of Internet Path Faults on Reactive Routing, ACM SIGMETRICS Performance Evaluation Review, Vol. 31, N. 1, 2003.

[FRHB13] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. IETF RFC 6824. Jan. 2013.

[FKC11] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0 (RFC 6101), 2011.

[GSG13] P. Gill, M. Schapira, and S. Goldberg. A Survey of Interdomain Routing Policies. ACM SIGCOMM Computer Communication Review, Vol. 44, N. 1, pp. 28-34, 2013.

[HRE+14] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing forged SSL certificates in the wild. In Proceedings of the IEEE Symposium on Security and Privacy, pp. 83-97, 2014.

[HRK+12] R. Holz, T. Riedmaier, N. Kammenhuber, and G. Carle. X. 509 Forensics: Detecting and Localising the SSL/TLS Men-in-the-middle. In Computer Security–ESORICS, pp. 217-234, 2012.

[KAF+10] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thom_e, 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. LNCS 6223, pp. 333-350, 2010.

[KFR06] J. Karlin, S. Forrest, and J. Rexford. Pretty Good BGP: Improving BGP by Cautiously Adopting Routes. Proceedings of the 14th IEEE International Conference on Network Protocols (ICNP), 2006.

[KS05] S. Kent and K. Seo. Security Architecture for the Internet Protocol (RFC 4301), 2005.

[L14] A. Langley. The POODLE bites again. https://www.imperialviolet.org/2014/12/08/poodleagain.html, Dec. 2014.

[L98] S. Lucks. Attacking Triple Encryption. In Proceedings of the 5th International Workshop in Fast Software Encryption (FSE), pp. 239-253, 1998.

[M79] R. C. Merkle. Secrecy, Authentication, and Public Key Systems. PhD thesis, Stanford, CA, USA, 1979.

[MDK14] B. Möller, T. Duong, and K. Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback. Security Advisory. Goole. Sep. 2014.

[MH81] R. Merkle and M. Hellman. On the Security of Multiple Encryption. Communications of the ACM, Vol. 24, N. 7, pp. 465-467, 1981.

[ML14] B. Möller and A. Langley. TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks (DRAFT). https://tools.ietf.org/html/draft-ietf-tls-downgrade-scsv-00, 2014.

[MOV96] A. Menezes, P. van Oorschot, and S. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[NIST12] NIST. Recommendation for Key Management - Part 1: General. NIST Special Publication 800-57, Revision 3(July):1-147, 2012.

[PS03] V.N. Padmanabhan and D.R. Simon. Secure Traceroute to Detect Faulty or Malicious Routing. ACM SIGCOMM Computer Communication Review, Vol. 33, N. 1, pp. 77-82, 2013.

[R92] R. Rivest. The MD5 Message-Digest Algorithm (RFC 1321), 1992.

[RLH06] Y. Rekhter, T. Li, S. Hares: A Border Gateway Protocol 4 (RFC 4271). Jan. 2006.

[RS92] C. Rackoff and D. Simon. Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack. Advances in Cryptology (CRYPTO), pp. 433-444, 1992.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, Vol. 21, N. 2, pp. 120–126, 1978.

[S15] B. Schneier. SHA-1 Freestart Collision. https://www.schneier.com/blog/archives/2015/10/sha-1_freestart.html, 2015.

[S90] W. R. Stevens. Unix Network Programming. Prentice Hall, New York, NY, 1990.

[SCB13] Schlamp, J., Carle, G., Biersack, E.W. A Forensic Case Study on as Hijacking: the Attacker's Perspective. ACM SIGCOMM Computer Communication Review, Vol. 43, N. 2, pp. 5-12, 2013.

[SHS15] Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS) (RFC 7457), 2015.

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

[SP10] E. Stavrou and A. Pitsillides. A Survey on Secure Multipath Routing Protocols in WSNs. Computer Networks, 54(13):2215-2238, 2010.

[Ss12] B. Schneier. When Will We See Collisions for SHA-1?, 2012.

[S95] 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.

[St12] M. Stevens. Attacks on Hash Functions and Applications. PhD thesis, 2012.

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

[V15] F. Valsorda. Logjam: the latest TLS vulnerability explained. https://blog.cloudare.com/logjam-the-latest-tls-vulnerability-explained/, 2015.

[VCB+13] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, P. Verissimo. Efficient Byzantine Fault Tolerance. IEEE Transactions on Computers, vol. 62, n. 1, pp. 16-30, Jan. 2013.

[WY05] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT), pp. 19–35, 2005.

[YL06] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture (RFC 4251), 2006.

[ZSR02] L. Zhou, F. B. Schneider, and R. Van Renesse. COCA: A Secure Distributed Online Certification Authority. ACM Transactions on Computer Systems Vol. 20, N. 4, 2002.