

Report

Software Engineering and Computer Science

Improving Openstack Tacker's Accountability Using Blockchain Technology

Written by

Jebbar Yassine

Supervised by

Dupont Sébastien

July 2017



**CENTRE D'EXCELLENCE EN TECHNOLOGIES DE
L'INFORMATION ET DE LA COMMUNICATION**

Charleroi, Belgium

Abstract

The emergence of Cloud Computing and Virtualization gives new opportunities and solutions in terms of flexibility and services abstraction. In networking, the abstraction of services has been mainly shown through improving the networks programmability (SDN) and decreasing the dependence between networking functions (DNS, NAT,...) and the specific underlying hardware (NFV). Service Function Chaining (SFC) is a main breakthrough of services abstraction in networking. It presents networking services as an ordered set of basic networking functions, hence the increased flexibility. Nevertheless, such architecture would require improving the security aspect in order to ensure the networking functions are working as expected. In this work, we look into Blockchain, the technology laying behind the famous digital currency Bitcoin, and its potential impact on improving SFC security. Firstly, we define the Service Function Chaining and explain the different concepts related to it, then we analyse the various security aspects that can be improved in these networking architectures. As we consider mainly a blockchain-based solution for SFC security, we benchmark many blockchain solutions for this use case, namely, but not exclusively, Namecoin, Blockstack and Hyperledger, which allows us, eventually, to choose Bitcoin's Blockchain as the most convenient solution for our problem space.

Keywords: SFC, NFV, Blockchain, Service Functions, Accountability.

Remerciement

Je tiens à remercier M. Mahmaz Youssef qui m'a accueilli et qui a accepté de m'encadrer aussi attentivement, et qui m'a encouragé et aidé tout au long de ce projet. Aussi, je m'acquitte volontier d'un devoir de gratitude envers lui, pour ses remarques pertinentes qui m'ont été d'une aide précieuse et pour sa disponibilité lors de ma réalisation du travail demandé, ainsi pour son soutien et sa compréhension. Je lui accorde aussi une grande gratitude pour tous les acquis dont il m'a doté, et qui ont constitué la base de tout progrès atteint lors de ce projet. Un grand merci également à toute personne ayant contribué de près ou de loin à l'accomplissement de mon travail.

Jebbar Yassine

Septembre, 2015

Institut National des Postes et de Télécommunication.

Contents

Abstract	i
Acknowledgement	ii
1 Introduction	1
2 Network Functions' Virtualization	3
3 Software Defined Networks	4
4 Service Function Chaining [1]	6
4.1 Definition	7
4.2 Components	7
4.3 Architecture	8
5 SFC Security Concerns Overview [2]	11
6 Blockchain Technology	14
6.1 Digital Transactions' Issues [20]	14
6.2 How Does Blockchain Overcome Double Spending ?	14
7 Problem Domain	19
7.1 Authentication	19
7.2 Authorization	20
7.3 Accountability	20
7.4 Conclusion	20
8 Blockchain-Based Solutions For Records-Keeping	22
8.1 Blockchain Platforms	22
8.2 Blockchain-based record keeping solutions	23
9 Blockchain Record-Keeping with OP_RETURN	27
9.1 Record-Keeping using Bitcoin's Blockchain	27
9.2 OP_RETURN field overview[26]	28
9.3 Constraints	30
10 Openstack Tacker	32
10.1 NFV MANO Framework	32
10.2 Does Tacker implement NFV MANO ?	34
11 Analysis & Conception	35
11.1 Problem Space Considerations	35
11.2 Logs Immutabilization	35
11.3 Immutable Logs Mechanisms [36]	36
11.4 Components	37
12 Implementation	41
12.1 Technologies	41
12.2 Components Implementation	42
12.3 Design Decisions	45
13 Conclusion	45
References	46

List of Figures

1	Network Functions Virtualization paradigm	3
2	Software Defined Networks architecture	5
3	2 different SFPs in a service function chain of 4 VNFs	8
4	SFC control plane and data plane interaction	10
5	The forming of a Merkle Tree in the Blockchain	15
6	Construction of the Blockchain	16
7	Transactions' chaining in the Blockchain	17
8	The transaction's information extracted from Bitcoin Core	29
9	Checking the transaction's different fields on blocktrail.com ...	29
10	OP_RETURN message	30
11	Translation of the hexadecimal OP_RETURN message	30
12	the NFV MANO framework architecture	33
13	Openstack's Tacker architecture	34
14	Logs' immutabilization and validation processes' components diagram	37
15	The publisher subcomponents	38
16	The validator subcomponents	39
17	The sequence diagram of the immutabilization process	40
18	The line hashe concatenated at the end of each logs entry	43
19	The checkpoint hashes read from the secure logs and stored in a new file	43
20	Bitcoin transactions sent once an hour at 12:58, 13:58 and 14:58	44
21	The stored transaction ids	44
22	The client python script reading the validation message from the queue	44

Introduction

In large scale networks, such as in datacenters, the conception and setup of the network topology is usually challenging, bearing in mind not only the careful choice of software/hardware that must be done to satisfy accurately the specific requirements, but also how a compatibility among the hardware devices provided by different vendors can be reached. Nevertheless, this variety of hardware vendors could create further problems whenever a change in the network's topology or further networking functions must be added in the datacenter. On the other hand, the traditional approach of networking has always been about the hardware, given that networking devices are offered to customers pre-built to work in a specific way, hence offering minimum flexibility to control and manipulate the technologies and protocols laying upon the hardware devices, depending on the customer's and network's situation. These two major limitations can often prove costly in terms of network's performance, or at least can prevent major performance improvement in some situations. There is no better way to illustrate those potential improvements than an example. In a simple topology where many hosts are connecting to a router (modern router that can do also the switch role), let's imagine a situation, where, for the sake of the argument, only two hosts are currently active, and have no interest in connecting to an external network, and want instead to exchange files between them using FTP. With the provided router, only a pre-determined range of the bandwidth can be given for these 2 hosts and for the FTP protocol as well. In an ideal situation, the majority of the bandwidth would be given to these 2 hosts and for the FTP exchange, since no other host is currently active, and also because the 2 hosts do not want to connect to an external network simultaneously. Hence the introduction of a new networking concept, SDN, where a (new type of) networking device can be controlled and manipulated independently from the technology that lays upon it, in order to improve the performance and increase the flexibility of the network. Obviously, the access to these devices and controlling them has to be done through a programming interface, and following some protocols, which explains the network being defined by software (and code). These details will be discussed in a later section. In spite of the flexibility that SDN can offer, the variety of hardware devices for the different networking functions (Firewalls, DNS,...) is proving to be resource consuming, especially in terms of energy, resulting therefore in higher costs. Another concept was introduced to resolve this limitation, Network Functions Virtualization (NFV), which is basically about allowing different networking functions to run on a common hardware platform, economizing as a consequence consumption

costs significantly, and prove to be useful in the network's topology complexity as well. These major concepts (SDN and NFV) and how they coexist to improve different aspects of networking will be discussed in more details in the next sections.

Network Functions' Virtualization

The new solutions and options that came with the cloud computing and virtualization emergence were so efficient in terms of performance and costs, that it was hard to let the opportunity of introducing such concepts to the networking field pass by, bearing in mind the profits that can be made in terms of cost and energy in that regard. Given that the virtualization concept is about maximizing the software/hardware independance, the main challenge in virtualizing networks was to provide a common hardware platform that can support running any networking technology upon it, and more importantly, that can also provide different networking functions, such as DNS and Firewalls. Further problems were encountered because of the different pieces of hardware provided by different vendors, which caused issues mainly related to the different drivers required for each device, and whether these devices can coexist with devices that are provided by a different vendor. The figure below (Figure 1) illustrates more clearly the NFV paradigm, where the networking functions are provided as an isolated entity from the infrastructure. The NFV management and orchestration component is serving as the linking component between the networking functions and the infrastructure, as it allows allocating physical resources (compute, storage, network,...etc) to the various virtual networking functions.

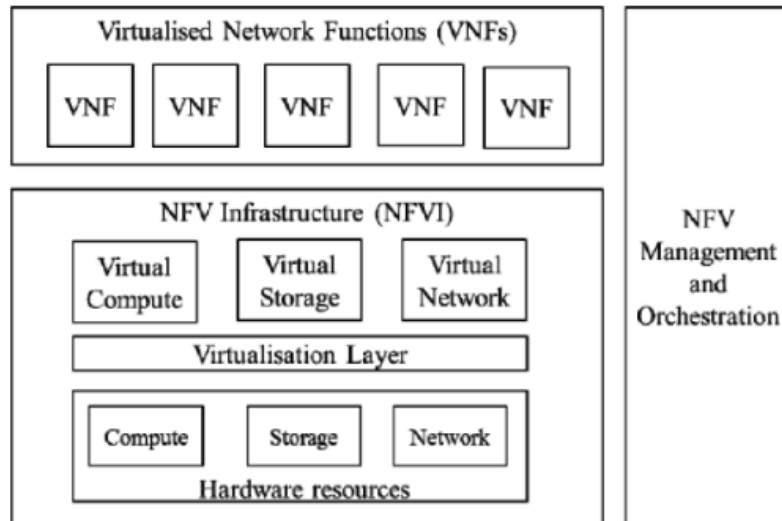


Figure 1: Network Functions Virtualization paradigm

Beside the hardware's complexity, problems emerged also on whether you can find IT experts to configure your datacenter, given that each device requires its own expertise. All these problems lead eventually to higher costs in terms of time, maintenance, and money. With NFV, not only the complexity was reduced, but it ensured that it was sufficient to connect the networking device to a simple computing device (laptop, server, workstation,...etc) to control it in the most simplistic way, with no need to the specific's device manual to configure it, which meant that even "I need a Cisco expert to configure this device" issue did no longer exist.

Software Defined Networks

The networking approach consisting of relying majorly on specific hardware, although efficient to some extent, has its fair share of limitations. One of the main disadvantages of this approach is the lack of flexibility to adapt the available bandwidth and the networking functions as required by the end user. In fact, the devices offered to customers leave barely the option to be functionnally controlled, and can only be accessed for configuration purposes. As a matter of fact, the networking equipments that are being commonly used by typical customers are pre configured. That is to say, that the details related to the bandwidth assigned to each protocol, especially if they are operating simultaneously (as that is usually the case), are defined by the vendor. For example, a salesman want to upload big files of the new automobile offers available in the market. At the same time, he has to send emails to his different customers to inform them about these new offers. Ideally, his networking device (router) should allocate all his available bandwidth to the HTTP protocol required for uploading his files on the web and the SMTP protocol to send mails, with a big bandwidth portion assigned to the HTTP protocol. Given that the networking device is already configured by the vendor, a significant portion of the bandwidth is being wasted on protocols that the salesman is not going to use for his operations (FTP, Telnet,...etc). Therefore, it was of a prime importance to provide a solution where the hardware can be manipulated flexibly, especially in large scale and complex networks, where the change of needs and maintenance emerge very frequently. Consequently, a new networking structure was suggested, where the hardware -The control plane- would only be needed for circuiting purposes and forwarding packets, while the forwarding plane (the controller), typically a server, is the 'brain' of the network, that is to say, the part of the network responsible for routing and every software-related aspect of the network. As illustrated in the figure below, the control plane, which consists

of networking protocols and generally the various software technologies involved in forwarding packets in a network, is abstracted from the hardware platforms that actually does the physical forwarding of data. The management plane is the interface allowing the communication between the control plane and the forwarding plane.

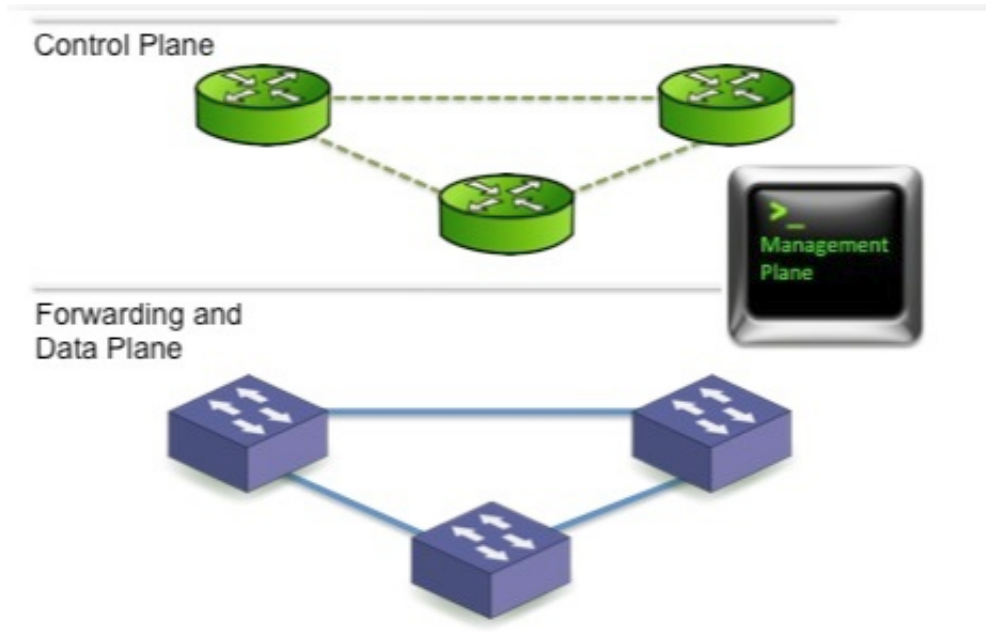


Figure 2: Software Defined Networks architecture

Given this new architecture, even large scale networks can be totally and flexibly controlled through programming interfaces. This approach does certainly offer numerous advantages as we can list below:

Direct Programmability As previously explained, decoupling the control plane from the forwarding plane allows direct access to the network. Therefore, the forwarding plane becomes programmable.

Agility Abstracting the technology from the hardware makes the network dynamically adjustable and adaptable to the needs of the network's administrators and users.

Central Management The centralization of the network's intelligence in the controller, which makes the whole network manageable from computing devices.

Vendors-Neutral Managing the network through servers increase the independence of the network from the hardware and the vendors that provide them.

Service Function Chaining [1]

We explained how advantageous both SDN and NFV can be for any network, with the former being more profitable in terms of software and technology, hence increasing the network's flexibility, while the latter offering hardware-related advantages, given how it provides common hardware platform that supports many networking functions laying upon it. However, network architects are always under obligation to combine all the networking solutions available in order not only to satisfy the requirements included in the customer's specifications, but to accomplish the most balanced specification/cost ratio possible as well. The SDN/NFV combination can massively contribute in reaching this balance. As explained before, SDN can increase the dynamic aspect of the network, thus improving its performance and maintenance. On the other hand, by providing common hardware platform that is able to support many networking functions, hardware and energy costs are significantly reduced. Eventually, this combination produces:

Low Complexity Given that the datacenter will contain the same common hardware for different networking functions, hence less vendor lock-in issues.

High Maintenance Ability Another direct consequence of lack of vendor lock-in, as the need for specific devices expertise will no longer be required.

Improved Performance Adaptability the abstraction of the technology from the hardware offers improved flexibility and adaptability to the end user requirements. It does also enhance the performance, especially when complicated (i.e. high bandwidth) operations need to be done over the network.

More importantly, the combination of these two fundamental concepts gave birth to a new form of networking, where network functions are presented as an ordered set (chain) of basic service functions, increasing therefore the flexibility of the networking functions and their adaptability to the network's changing requirements. In the next parts of this work, we define Service Function Chains as well as its main components and how they interact.

Definition

SFC can be defined as a concept that enables the creation of composite networking services (Service Functions) that are strictly ordered, in order to apply them to packets/frames matching the rules of these Service Functions (Classification). SFC also offers a method of deployment for SFs, in which the order of the service functions can be dynamically changed, and do not depend on the underlying topology, using simply the exchanged messages.

Components

Service Function A specific function that processes incoming packets following predefined rules. Functions can be involved in the delivery of added-value services. A non-exhaustive list of abstract service functions includes: firewalls, Deep Packet Inspection, NAT...

Service Function Forwarder (SFF) A logical component that sends packets to the connected service functions according to information carried in the SFC encapsulation. It does also handle traffic coming back from the Service Functions.

SFC Metadata Exchanged context information among the different SFC components.

Service Function Path (SFP) a constrained specification of where packets assigned to a certain service function path must go. Specific details declaration is not mandatory. While in some cases the different Service Functions to be visited are fully specified, we can always maintain a certain level of abstraction in some other cases. In the figure below, we can see 2 different service function paths, where the packets forwarded by first one (green line) goes through the firewall virtual network function and then pass through the video optimizer VNF. Meanwhile, the second SFP consists of packets going by the firewall VNF, then they are forwarded to the intrusion and malware detection VNF, and finally the parental control VNF is applied upon them.

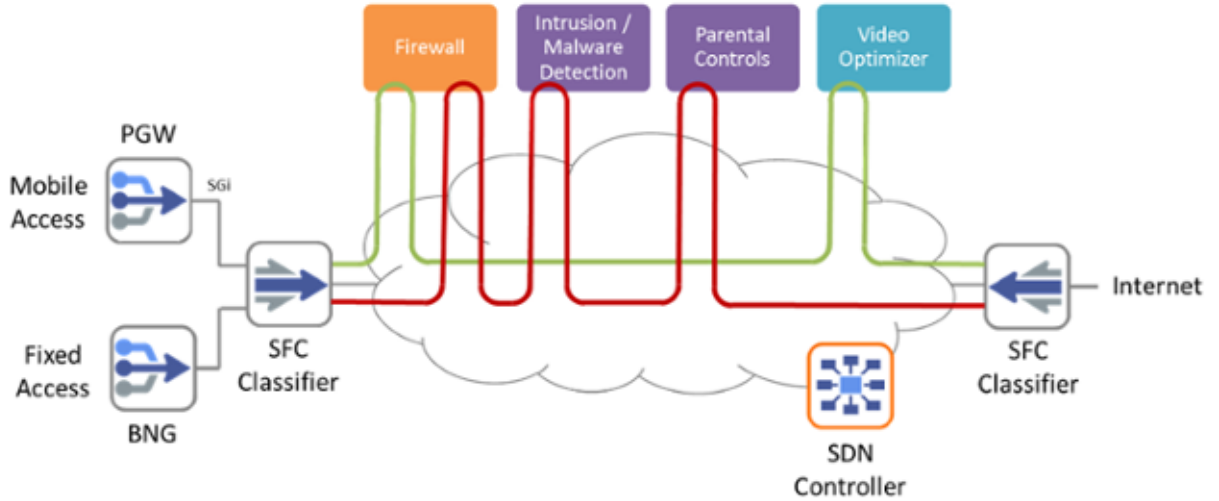


Figure 3: 2 different SFPs in a service function chain of 4 VNFs

SFC Encapsulation The SFC encapsulation provides identification for SFP. It is also used by the SFC-aware functions. In addition, the SFC encapsulation carries metadata.

Rendered Service Path The different SFs and SFPs visited by a packet in the network.

SFC-Enabled Domain A network or region of a network where SFC is implemented.

SFC Proxy Logical component that removes and inserts SFC encapsulation on behalf of an SFC-unaware service function.

Architecture

A SFC architecture does usually have to satisfy a set of principles in order to ensure efficient SFC usage.

Topological Independence The deployment of SFs or SFCs does not require any changes in the underlying network forwarding topology.

Plane Separation The packet handling operations should not be confused

with the dynamic realization of the Service Forwarding Paths. For example, if a packet has to be forwarded through 2 VNFs, as the SFF chooses the convenient (available) instances for these 2 VNFs, the outcome of the operations applied upon the packet should be similar whatever the instances chosen for these 2 networking functions.

Classification A specific SFP is responsible for any packet matching a specific rule.

Shared Metadata Metadata/context data can be shared amongst Service Functions and classifiers, between SFs, and between external systems and SFs. Metadata could be used to provide and share the result of classification (that occurs within the SFC-enabled domain, or external to it) along a SFP.

Service Definition Independence The SFC architecture does not depend on the details of SFs themselves.

SFC Independence The creation, modification, or deletion of a SFC has no impact on other SFCs. The same is true for SFPs.

Heterogeneous control/policy Points The architecture allows SFs to use independent mechanisms to populate and resolve local policy and local classification criteria.

These architecture principles should be applied to the different planes composing a SFC network. The environment associated to SFC is mainly separated into four main planes:

The Control Plane mainly responsible for controlling and configuring the SFC related components. It should be noted that the control plane does only concern a subset of the parameters and facilities associated to the SF.

The Management Plane its main role is to allocate resources to the various SFs and eventually activate the various SFs components. Management operations define the number of CPUs and memory bandwidth associated to the SFs, as well as specific configuration parameters of the SFC components. Unlike the control plane, the management plane defines a large number of parameters related to the SFC components configuration.

The Data Plane consists in all SF components as well as the data exchanged between the SF components. Communications between SF components includes the packet themselves, their associated metadata, the routing

logic or SF logic. The SFC Data Plane can also be seen as all the elements that interact with a packet provided by an end user.

The SFC Tenant's Users Data Plane consists in the traffic data provided by the different users of the tenants. When a user is communicating with a server or another user, typically from another administrative domain, the communication belongs to the SFC Tenant's Users Data Plane whenever packets are provided by the server or by the user.

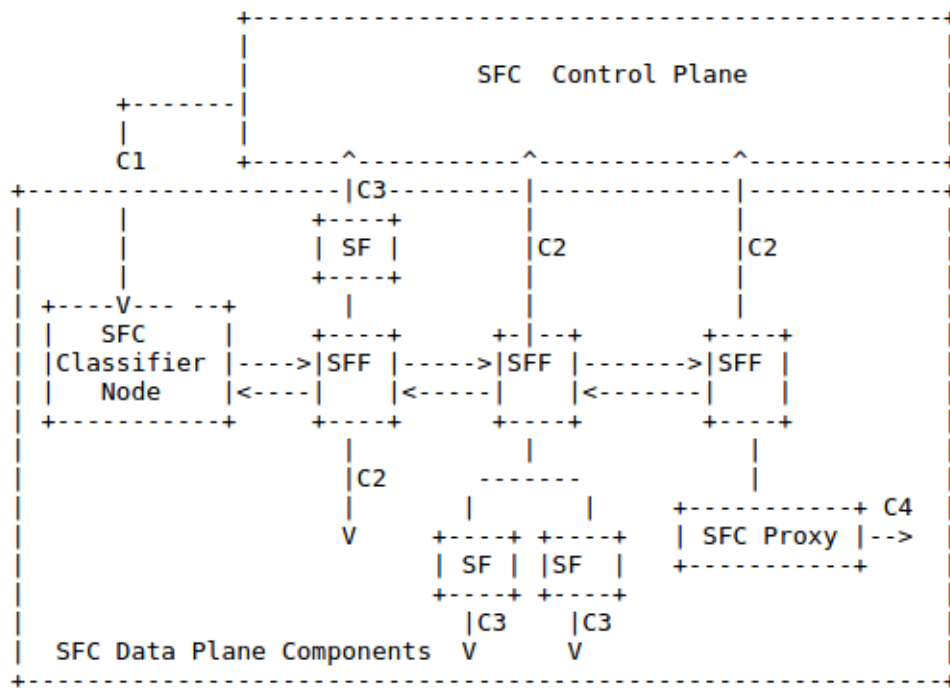


Figure 4: SFC control plane and data plane interaction

SFC Security Concerns Overview [2]

The essential infrastructure required for the deployment of SFC is usually provided by a Cloud Provider. This infrastructure does include dedicated hardware serving a specified network function. The network function served by the dedicated hardware is the actual Service Function. If the SFC domain is not private to the company, the infrastructure and the dedicated hardware that it contains is often shared by multiple tenants. In some other cases, a local proxy transparently redirects the local SFC network to an external SFC domain outside the local boundaries of the local infrastructure. Each SFC Tenant is responsible for its domain, that is to administrate or provision the necessary resources and control all its SFC elements (e.g: defining SFC Paths, configuring the elements,...etc). A SDN controller is responsible for the coordination of the SFC elements.

As for the security requirements for an SFC domain, they aim to protect the deployed SFC architecture from attacks. Even in a private SFC deployment, where SFC components are considered to be in a trusted environment, inside attacks are still possible (e.g. inside attacker sniffing the SFC metadata, sending spoofed packets,...etc). The evolution of the local architecture could require, at some point, interconnecting with a third party SF/SFF, which puts the initial domain basically outside of the local (private) domain. Multitenancy also represents a security concern, due to the fact of sharing a SFC platform. Unless the tenants are strongly isolated (physically or logically), different networks may share a common SFF, and one tenant may update the SFP of the other tenant. Such misconfiguration has similar impact as a redirecting attack.

The threats in this work are analysed in each plane. In fact, although the SFC architecture is divided into 4 planes, in order to control and limit the interactions among the different SFC components, but these interactions still exist and so may be used by an attacker. As a result, for each plane, the threat analysis is performed by analysing the vulnerabilities present within each plane, as well as those performed via the other planes. We focus mainly on the threats faced by the Data Plane.

Attacks may be performed from inside the SFC Data Plane or from outside the SFC Data plane. Therefore, the attacker is in at least one of the following planes: SFC Control Plane, SFC Management Plane or SFC Tenants' Users Plane.

1. Attacks performed from outside the SFC Data Plane

Attacks performed from the SFC Control Plane

Vulnerabilities can be found basically in the interfaces used for communication between the SFC Control Plane and the SFC Data Plane. These interfaces are responsible for updating the classification rules for the SFC classifier, updating forwarding decisions for SFFs and updating SFs/SFC proxies internal state. An attacker may change the SFC Classifier classification and completely modify the services provided by the SFC. This could result in losing control over the tenant's traffic.

Attacks performed from the SFC Management Plane

This type of attacks is similar to the previous type, with the only difference being that the SFC Management Plane usually provides a greater control over the SFC component than the SFC Control Plane.

2. Attacks performed from inside the SFC Data Plane

Given that an attacker has taken control of a SFC component, various types of attacks can be performed, such as modification of the traffic, performing onpath attacks (i.e: modification of packet content as it is forwarded by components), generating additional traffic to create heavy load situations. However, The traffic within the SFC Data Plane is composed of multiple layers: the transport layer, the SFC encapsulation layer and the SFC payload layer. As a result, attacker may use the traffic to perform attacks at various layers.

Attacks performed at the transport layer

Mainly related to the illegitimate SFC traffic that could be provided to the SF. A malicious node that is not expected to communicate with that SF may inject packets into the SFC, That may eventually spoof the IP address of legitimate SF, so the receiving SF may not be able to detect the packet is not legitimate.

Attacks performed at the SFC encapsulation/payload layer

The SFC encapsulation and payload are considered as SF inputs. Therefore, attacks can be performed through them. Injecting malicious metadata

in the encapsulation envelope may allow to inject traffic, due to the fact of escaping traffic authentication. When SFC traffic is not authenticated, an attacker may also modify on-path the packet. By changing some metadata contained in the SFC Encapsulation, the attacker may test and discover the logic of the SFF. Similarly, when the attacker is aware of the logic of a SFC component, the attacker may modify some metadata in order to modify the expected operation of the SFC.

Blockchain Technology

Blockchain has made a name for itself in the IT field as being the technology behind the digital currency Bitcoin [20]. Nevertheless, it is quickly being expanded to other fields (Health Care, Networking Security,..etc) after proving its worth in securing digital systems robustly. In this work, we take a look at the Bitcoin's Blockchain model, and the mechanisms used to prove the authenticity of digital transactions without the involvement of a trusted third-party.

Digital Transactions' Issues [20]

In traditional face to face transactions, the money-asset exchange would usually take its course smoothly, with both the seller and the buyer confronting each other and witnessing the trading operation with their bare eyes. However, the anonymity of digital transactions has left many points of exploitation and manipulation, as the parts involved in these kind of transactions need to trust the digital system in order to get, eventually, their part of the deal. Double spending is arguably the most obvious manipulation scenario in digital transactions, where, either the buyer or the seller, would use the same money (or the same asset) in different trading deals with different parts. The solution to this issue that has been in place for many years is the involvement of a third "trust" party, that would ensure the authenticity of the deal and that neither the buyer nor the seller are being actually manipulated. This trust party, which is usually a central bank, would also demand a percentage from the deal for its service. The emergence of the Blockchain though, has introduced a whole new paradigm to solve the trust issue. Details of this new paradigm and how it solves the double spending problem are discussed in the next subsection.

How Does Blockchain Overcome Double Spending ?

Since we have no interest in the financial aspect of the Bitcoin, this subsection will contain merely a technical explanation of how Blockchain accomplishes the authenticity of transactions from a technical and computing perspective, without giving any details about the fees generated in these transactions and the extra fees that are received by the network's different participants (see [20] for more details). We do also justify skipping the financial details in our work by the fact that our implementation will use tBC [29] (testnet BTCs) instead of real Bitcoins (ie: mainnet BTCs). The Bitcoin's testnet has been mainly created for development reasons and testing different use

cases using the Bitcoin's Blockchain, profiting simultaneously from a similar network to the main Bitcoin's Blockchain and without being costly for IT developers, as the tBTC has no real value and you can get them for free online (see [30]). Therefore, real transactions (similar to mainnet's transactions) that take place in the testnet are used for different development scenarios by Blockchain developers basically for free. In the next sections of this work, we walk through the various mechanisms and techniques used by Bitcoin's distributed ledger technology to store the "hard to alterate" transactions' records.

1. Merkle Trees [11]

Merkle trees are binary trees of hashes. Merkle trees in bitcoin use a double SHA-256, the SHA-256 hash of the SHA-256 hash of something. When a row in the tree is being formed (other than the root of the tree), it would have an odd number of elements, the final double-hash is duplicated to ensure that the row has an even number of hashes. The bottom row of the tree is initially formed with the ordered double-SHA-256 hashes of the byte streams of the transactions in the block. Then, the row above it consists of half that number of hashes. Each entry is the double-SHA-256 of the 64-byte concatenation of the corresponding two hashes below it in the tree. This procedure repeats recursively until we reach a row consisting of just a single double-hash. This is the Merkle root of the tree. The figure below illustrates a merkle tree of a number of hash values. As shown in the example, each pair of (leaf) hash

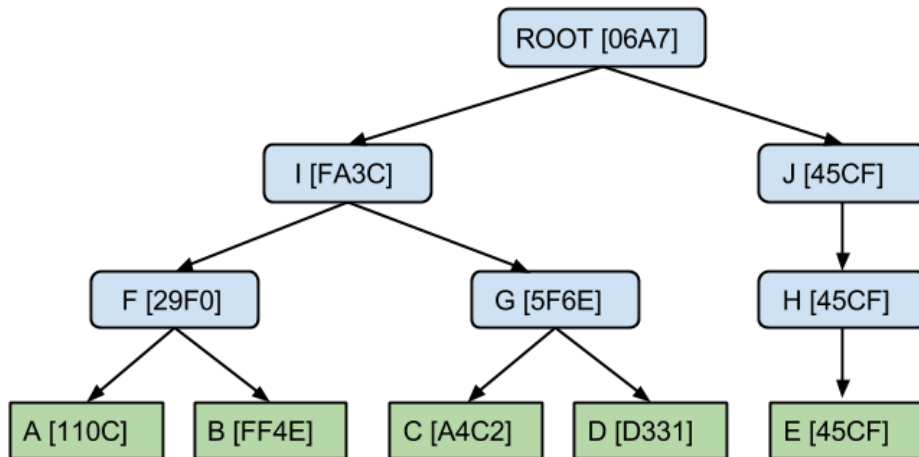


Figure 5: The forming of a Merkle Tree in the Blockchain

values are hashed together to construct a root hash. Eventually, a root of the tree is computed, which is the root of the merkle tree.

2. Nonce Computation (Proof of Work) [12]

The proof of work used in Bitcoin takes advantage of the "random" nature of cryptographic hashes. A good cryptographic hash algorithm converts arbitrary data into a seemingly-random number. If the data is modified in any way and the hash re-run, a new seemingly-random number is produced, so there is no way to modify the data to make the hash number predictable. To prove you did some extra work to create a block, you must create a hash of the block header which does not exceed a given value. This given value is a hash beginning with a predefined number of 0's in the case of Bitcoin's Blockchain . For example, if the maximum possible hash value is $2^{256} - 1$ (this maximum value possible is also known as the Blockchain's current target), you can prove that you tried up to two combinations by producing a hash value less than 2^{255} .

3. Blocks' Creation[12]

The root of the merkle tree is stored in the block header, which also contains the hash of the previous block header, to ensure that a modification in a block will also change the other blocks.

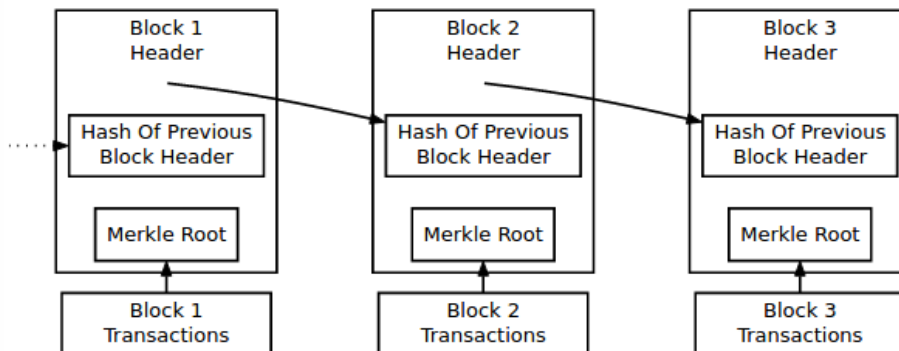


Figure 6: Construction of the Blockchain

Transactions are also chained together. Each transaction spends the satoshis (1 satoshi = 0.00000001 bitcoin) previously received in one or more earlier transactions, so the input of one transaction is the output of a previous transaction.

A single transaction can create multiple outputs, as would be the case when sending to multiple addresses, but each output of a particular transaction can only be used as an input once in the blockchain. Any subsequent reference is a forbidden double spend—an attempt to spend the same satoshis twice. Outputs are tied to transaction identifiers (TXIDs), which are the hashes

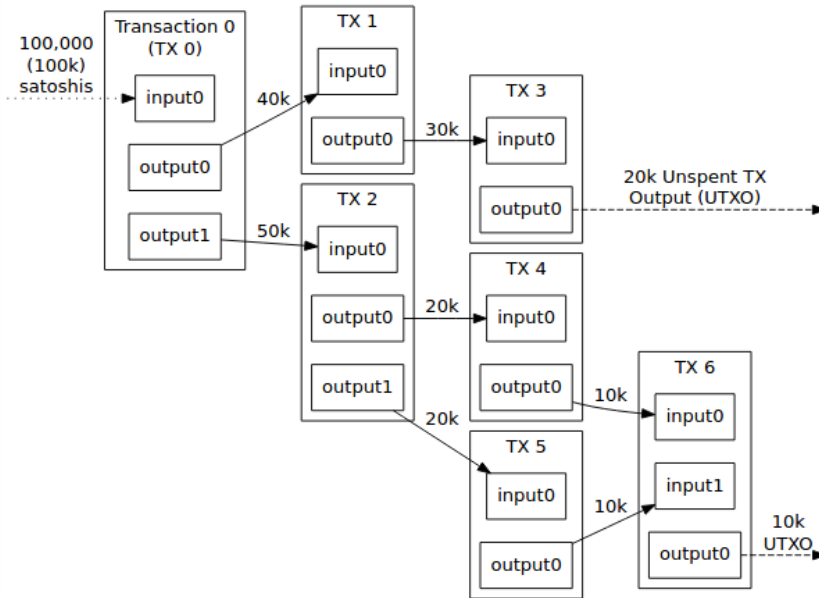


Figure 7: Transactions' chaining in the Blockchain

of signed transactions. Because each output of a particular transaction can only be spent once, the outputs of all transactions included in the block chain can be categorized as either Unspent Transaction Outputs (UTXOs) or spent transaction outputs. For a payment to be valid, it must only use UTXOs as inputs.

New blocks will only be added to the block chain if their hash is at least as challenging as a difficulty value expected by the consensus protocol. Every 2,016 blocks, the network uses timestamps stored in each block header to calculate the number of seconds elapsed between generation of the first and last of those last 2,016 blocks. The 2016 was arbitrary from the perspective that the level of difficulty should be checked every 2 weeks. Given the ideal value is 1,209,600 seconds (2016 blocks*10 minutes*60 seconds) (two weeks), if it took fewer than two weeks to generate the 2,016 blocks, the expected difficulty value is increased proportionally (by as much as 300%), so that the next 2,016 blocks should take exactly two weeks to generate if hashes are checked at the same rate. If it took more than two weeks to generate the

blocks, the expected difficulty value is decreased proportionally (by as much as 75%) for the same reason. Because each block header must hash to a value below the set difficulty, and because each block is linked to the block that preceded it, it requires (on average) as much hashing power to propagate a modified block as the entire Bitcoin network expended between the time the original block was created and the present time. Only if the majority of the network's hashing power is acquired, an attacker could reliably execute such a 51 percent attack against transaction history (although, it should be noted, that even less than 50% of the hashing power still has a good chance of performing such attacks).

Problem Domain

Although the previous section contained a specific SFC security approach, we've chosen to define the problem domain for our work from a general security perspective. This change of perspective was by no means a random choice of ours, but it was rather due to the fact that Service Function Chains applications are still under continuous development, which makes it even harder to choose a specific SFC use case for security, or even focusing on a SFC plane to secure. This new SFC development on the cloud market become obvious when we look at the "established" NFV platforms that are supposed to provide SFC services. For instance, Openstack, a major leader in the IaaS (Infrastructure as a Service) market, still doesn't give a "stable" solution in terms of SFC. In spite of the promising signs that Openstack is showing in "port chaining" in its networking component Neutron, and the current NFV platform main project Tacker, port chaining is more of a chain of classification than network function chain, as "Service VMs must be attached at points in the network and then traffic must be steered between these attachment points" [31]. On the other hand, Tacker, besides the fact of its limited documentation to basic applications, is still showing bugs when an adaptation of context is required for a specific use case (19 official bugs in 2017 as of the moment of writing of this report)[3]. For all these reasons, we discuss the problem domain from the security three A's perspective: Authentication, Authorization, Accountability. Authentication represents the ability to access to the system, be it from actual users, components, traffic, or any participating entity of the system. Authorization, however, specifies the functionalities that each entity has access to after getting authenticated by the system. Finally, accountability provides the feature of tracing/keeping the records of the different actions that are being performed within the system, allowing thereby to refer each action to the system's participant that has performed it. Therefore, accountability prove its massive usefulness after the system has been breached, or if it's simply not working correctly. In this work, we give various proposals regarding how the Blockchain technology can help in securing these three aspects in the Service Function Chains context.

Authentication

Service Function Chains consists of interactions among various networking components, in order to decide eventually the path that the packets will follow and the service functions that will be applied upon them. Those interactions include the messages exchanged among the different components.

Authenticating these messages can increase the security of the chain and ensure that it is functioning properly. A hash of each message can be computed and stored in the blockchain. Components can check the blockchain after receiving a message, then they would simply compare the hashes to prove the authenticity of the message. A case for authentication in the blockchain can also be made for the interfaces, components and devices participating in the SFC.

Authorization

Defining different levels of authorization and privileges using blockchain is not as obvious as it may seem. A potential implementation of this feature using the blockchain technology may be done with the public and permissioned blockchains, in an architecture where the participants in the public blockchain may have only reading rights and consulting the contents of the chain, while another private/permissioned chain is made for the participants that can contribute in writing blocks within the chain.

Accountability

Using Blockchain for accountability is probably the most fitting use case in the SFC context. Blockchains are naturally used in Bitcoins to keep record of transactions. There is also various blockchain applications for copyrights [4], digital signatures and verification [5], and health care records [6]. If anything, it only shows how the blockchain is used for keeping immutable records in different contexts. Likewise, blockchain could be deployed in Service Function Chains to keep records and logs of what is performed inside the chain: exchange of messages, classification rules' update, change of paths for some Service Functions...etc. Blockchain will not ensure only keeping the records inside the chain, but it will guarantee their integrity as well, since those records are being witnessed by the multiple participants within the chain and stored in the distributed ledger, allowing thereby safe monitoring of the SFC and knowing exactly who did what.

Conclusion

In an ideal scenario, our work would have included the different potential use cases for securing Service Function Chains with Blockchain. However, this work is being accomplished during an internship, and is, consequently, limited by time constraints. Moreover, even if the future looks bright for Service Function Chains as well as Blockchain technology, they're still discussed

in theory rather than put in place as real life applications [7] [8]. Consequently, we've chosen to focus on accountability of the SFC using Blockchain technology, as keeping records is Blockchain's main use case. Although it was introduced initially for financial transactions' use, this work will require adapting the technology for the service function chains context.

Blockchain-Based Solutions For Records-Keeping

Blockchain can be generally described as a distributed ledger linking numerous blocks of data in a sequenced manner, in order to keep track of the different transactions taking place within the system. The content of the ledger is being agreed on by all the participants in the system, through a distributed consensus algorithm, in which every participant is witnessing the transactions taking place and therefore proposing a block of transactions based on these witnesses. Although the principle is the same, different blockchain platforms choose different approaches (public/private, PoW/PoS,...) to implement this technology. This approach variation is mainly due to the application field (Finance, Health care, Copyrights protection,...) and what a developer may want to achieve with the blockchain technology stack. Although a benchmarking of the record keeping solutions would have presented immense profit for our cause, the big diversity of these solutions in terms of approach, the platform they lay upon and the target problem domain made it even harder for us to find a stable criteria for comparing the solutions and find out to what extent they fit for our SFC use case. Therefore, the remaining parts of this section offers simply an overview of multiple blockchain solutions that are mainly destined to keep records in different contexts, with a display of their main characteristics and advantages.

Blockchain Platforms

1.Bitcoin

The first, and most famous blockchain platform. Launched in 2009 by the anonymous Satoshi Nakamoto, the blockchain technology was directly involved in making Bitcoin the most valued cryptocurrency up to date [13]. The Bitcoin blockchain maintains the transactions' records made by the different participants in the Bitcoin network. It does also rely on the proof of work principle to prove the authenticity of the records. PoW is basically about solving a mathematical puzzle consisting of finding the right nonce. The "nonce" in a bitcoin block is a 32-bit (4-byte) field whose value is set so that the hash of the block will contain a run of leading zeros. the number of zeros is global variable in the Bitcoin network, and it changes after every 2016 blocks [14]. Bitcoins are put into circulation by mining. Mining is "is the process of adding transaction records to Bitcoin's public ledger of past transactions or blockchain" [15].

2. Ethereum

Relying on the same blockchain principles, ether (Ethereum-based cryptocurrency) is emerging as the second most successful digital currency behind bitcoins [13]. Nevertheless, Ethereum was designed to be much more than a payment system. It is "a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third party interference" [16]. Ethereum block times are currently at about 14 seconds, compared to Bitcoin's 10 minutes. Ethereum also currently operates on a proof-of-work basis. Miners are rewarded for processing transactions and executing smart contracts, which create blocks.

Blockchain-based record keeping solutions

As interest in blockchain-based solutions -be it through companies or simple initiatives/projects- is growing dramatically [17], many companies who are adopting this technology prefer to build their solutions either on the Bitcoin blockchain or the Ethereum blockchain [18], given their well-established status as the two most successful Blockchain platforms currently [13]. Therefore, the evaluation of the record keeping solutions in these section should take into account how these solutions are making profit from Bitcoin/Ethereum Blockchains to ensure the security of their records.

Blockstack

Originally a naming system which is built on Bitcoin's Blockchain, Blockstack provides the ability of tracking key-value pairs in form of (unique) names and their associated data. In the Blockstack Whitepaper [19], there is another confirmation, after a handful of experiences, that the Bitcoin's Blockchain is probably the most secure Blockchain platform available currently. If that is partly due to the big network of participants in Bitcoin, the difficulty of controlling 51% of the processing power inside the network, where all the participants are even, is an important factor as well. On the other hand, Blockstack developers chose to take a different approach in terms of storage, given that the Bitcoin Blockchain can only hold data in the order of kilobytes [20]. Therefore, Blockstack uses blockchain "as a communication channel for announcing state changes, as any changes to the state of name-value pairs can only be announced in new blockchain blocks" [19]. As for storing these records of the name-data pairs, Blockstack uses a dedicated data plane for that purpose.

Proof of Existence [21]

An online service for storing and timestamping the existence of a document inside the blockchain. This secure storage is done traditionally by keeping a hash of the file in the chain, with a reference also to the time in which the document was stored. Consequently, such mechanism does not only ensure that the files' content cannot be seen by the Bitcoin's network participants (since PoE is built on Bitcoin's Blockchain), it allows also validating the document's existence "even if this site (proofofexistence.com) is compromised or down". On the technical aspect, it is worth noting that this solution also uses the OP_RETURN field to pass specific information with the transaction. To confirm a document's existence at a certain timestamped time, checking the OP_RETURN field in blockchain transactions for the SHA-256 hash of the document, prepended by the PoE marker bytes (0x444f4350524f46) should be sufficient. The existence of that transaction in the blockchain proves the existence of the document at the timestamped time.

Hyperledger

Even though we stated earlier that most current well-established solutions are either Bitcoin or Ethereum based, Hyperledger succeeded to make a name for itself as a smart contract blockchain platform, relying on a somewhat different approach. Initiated by The Linux Foundation, Hyperledger started a set of distributed ledger solutions based on different platforms and programming languages. Hyperledger is based on the expectation that there will be many blockchain networks, with each network ledger serving a different goal [22]. In this section, we present the Hyperledger Blockchain explorer, and then we take a keen interest on Hyperledger Fabric.

1. Hyperledger Blockchain Explorer

A blockchain explorer web application. It allows "to view/query blocks, transactions and associated data, network information (name, status, list of nodes), chain codes/transaction families (view/invoke/deploy/query) and any other relevant information stored in the ledger" [23]

2. Hyperledger Fabric

Hyperledger Fabric is simply "a permissioned blockchain platform aimed at business use. It is open-source and based on standards, runs user-defined

smart contracts, supports strong security and identity features, and uses a modular architecture with pluggable consensus protocols" [24]. It aims to advance blockchain technology by identifying and realizing a cross-industry open standard platform for distributed ledgers, which can transform the way business transactions are conducted globally. Generally, Hyperledger Fabric offers the major advantage of ensuring private transactions/contracts between single nodes in a network of nodes. This is made possible through providing private communication channels between participants. Therefore, any two (or more) participating nodes can share their own distributed ledger, containing their own private transactions, with the rest of the network being unaware of this private ledger. As for the participating peers in the network, the fabric distinguishes between two kinds of peers: A validating peer is a node on the network responsible for running consensus, validating transactions, and maintaining the ledger. On the other hand, a non-validating peer is a node that functions as a proxy to connect clients (issuing transactions) to validating peers. A non-validating peer does not execute transactions but it may verify them [24]. the smart contracts run by Hyperledger Fabric are written in Go language, and they are called chaincode. The transactions' operations are generally divided into 3 categories

1. **Deploy** Allows traditional deployment of smart contracts, as this operation installs the chaincode taken as its input in the peers of the network. After the deployment, the chaincode can be invoked by the different peers.
2. **Invoke** Provide the ability to invoke a transaction of a particular chaincode that has been installed earlier through a deploy transaction [24]. Depending on the arguments taken by the operation; the chaincode defines the type and executes the transaction, may read and write entries in its state accordingly, and indicates whether it succeeded or failed.
3. **Query** Returns an entry of the state directly from reading the peer's persistent state [24].

Namecoin [25]

Just like Blockstack, Namecoin is another blockchain based solution for namespaces' storage. Relying on forking the Bitcoin's blockchain (using Bitcoin's blockchain itself for specific purposes and additional functions), Namecoin allows secure storage of key value pairs, ensuring consequently the uniqueness of names inside the blockchain. On the other hand, the values associated to different namespaces aren't necessarily unique, as the same value

can exist within different namespaces. For the additionnal functions, Namecoin introduced three new operations: **NAME__NEW** which allows a client to choose a name (if it's not already taken), **NAME__FIRSTUPDATE** takes as input the output of the previous operation, and allows the client to associate initial values to the chosen name and **NAME__UPDATE** takes as input the output of **NAME__FIRSTUPDATE** and gives the client the possibility to change or insert new values in the name that he has chosen.

Blockchain Record-Keeping with OP_RETURN

After doing an overview of various blockchain-based solutions for record-keeping, we present, in this section, a detailed explanation of our chosen solution for securing Service Function Chains using Bitcoin's Blockchain. We focus on the accountability of SFC, relying on the OP_RETURN field of blockchain's transactions to transmit the different messages exchanged between the SFC components, ensuring therefore immutable logging of the interactions that occur inside a Service Function Chain.

Accountability, the third A in the famous security 3 A's (Authentication, Authorization and Accountability), is a major feature that should be taken into account in any implementation of a security system. Accountability's importance does not only lay in providing records of the different exchanges between the system's components, but it does also allow identifying points and sources of failure when the system stops working as expected or suffers a performance degradation. In the SFC context, a Service Function may shut down, a Service Function Forwarder might equally struggle with forwarding packets due to an overload. Therefore, keeping records of the interactions happening within the system ensures the system is working as expected as well as its high availability. In the next section, we give a brief reminder of the blockchain's principle, then we explain in details how Bitcoin's blockchain can be used for record-keeping.

Record-Keeping using Bitcoin's Blockchain

Blockchain can be generally described as a distributed ledger that serves as a record keeper for transactions occurring among the different Bitcoin's network participants. By keeping records of these transactions, the peer to peer system ensures that assets (bitcoins) are not being double-spent. The general consensus protocol adopted by the blockchain allows the majority of participants to testify whether a transactions has effectively taken place or not. Eventually, a block of multiple transactions is confirmed every 10 minutes. Once a block get "chained" within the distributed ledger, it can never be changed or altered, or at least it will require taking down the majority of the computing power of the participants to change a confirmed block of the chain (51% attack). Given this strength of the blockchain of keeping records and ensuring their authenticity, we figured out a use case where the Blockchain can be deployed in the SFC context to keep records of the interactions occurring among the SFC components, and the messages that are being exchanged within the Function Chain.

OP_RETURN field overview[26]

In the early days of the Bitcoin, users and participants were not concerned how the underlying blockchain technology could be used in different contexts and for various use cases, as they were embracing the breakthrough of the digital currency emergence. As it started to get popular and gained momentum, IT developers proposed applications for the blockchain outside the finance industry: Keeping records for healthcare, Legal possession of assets, Copyrights...etc. Such applications would usually require sending/receiving different kind of messages with the transaction (confirmation, acknowledgement of receipt among others). As a consequence, developers used one way or another to get the messages sent with the transaction. Back in 2013 different players in the bitcoin ecosystem were trying to include bits of information into transactions so that they could take advantage of the irreversibility of the blockchain. Imagine, for instance, you wanted to write a contract and place it in an unchangeable location that at any future date one could go back to verify it existed. You can do this by using the blockchain. You add some bits to the transaction's scriptSig value that don't alter the end result of running that script, but allow you to store information like "I hereby declare to give asset A to address XYZ at time UNIX_TIMESTAMP". There were even stranger ways people would add extra bits, like including it in the BTC value of an output. Some members of the community did not like this, as they saw these extra bits as polluting the blockchain. The extra bits were a network efficiency concern because more bits meant larger block chains and more of an onus on those who are running full nodes, and it was also a community consensus concern because they thought "we all implicitly agreed to store financial data in the blockchain, which is important to everyone, but we did not agree to store data like small text messages and invoice text". To reach a middle ground in these opposing views, the core-developers made the opcode OP_RETURN a valid opcode to be used in a bitcoin transaction, which allows 80 arbitrary bytes to be used in an unspendable transaction. Later in February 2014, the bytes count was reduced from 80 to 40 bytes. On 16 Nov 2014 there was a pull request to have the OP_RETURN size increased "back" to 80. This was then accepted (merged) on 4 Feb 2015 [27]. To gain deeper knowledge about OP_RETURN transactions in practice, we made a transaction in Bitcoin's testnet containing "Hello, testnet!" in its OP_RETURN field. We've use the online Bitcoin's explorer <https://www.blocktrail.com>, which allowed to check the transaction's fields once it got confirmed inside a Bitcoin's block. The figures below show the transaction's details in Bitcoin's core and its fields in the Blockchain's explorer.

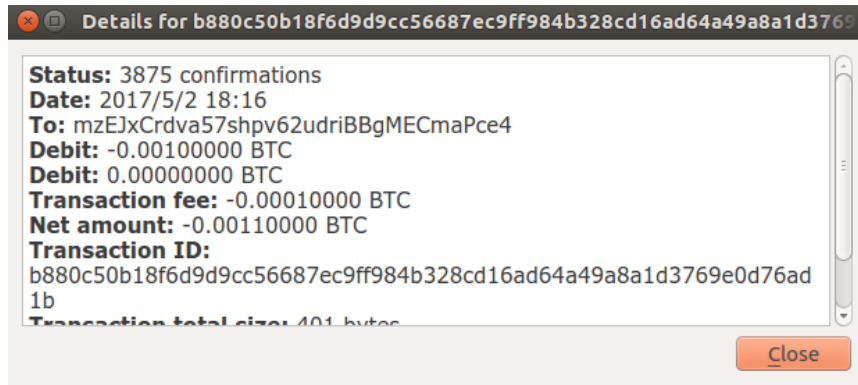


Figure 8: The transaction's information extracted from Bitcoin Core

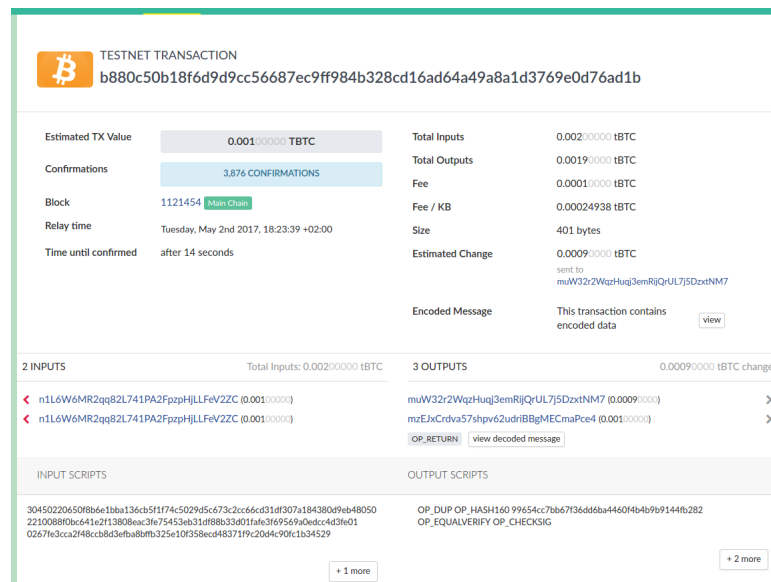


Figure 9: Checking the transaction's different fields on blocktrail.com

In the figure below (Figure 10), on top there is the hex message contained in the OP_RETURN field of the transaction. The first hex value (0x6a) is the OP_RETURN opcode in Bitcoin's stack-based programming language [28]. The second hex value (0x0f) refers to the number of bytes of the actual message.

```
6a0f48656c6c6f2c207465737446e657421
Hello, testnet!
```

Figure 10: OP_RETURN message

Therefore, our message only used 15 bytes (0x0f equals 15) of the 80 bytes offered by the OP_RETURN field. As shown in the figure below, translating the remaining part of the hexadecimal message to ASCII language results in the expected "Hello, tesnet!" message.

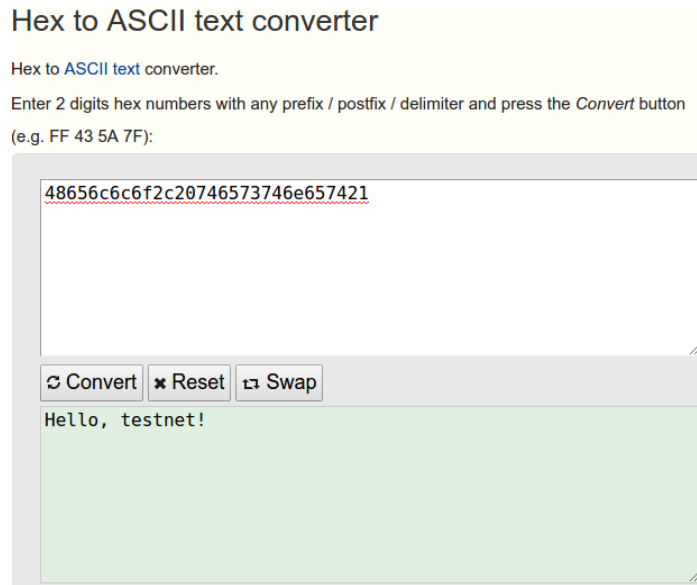


Figure 11: Translation of the hexadecimal OP_RETURN message

Constraints

Even though using OP_RETURN for transmitting (and eventually storing) the exchanged messages inside a Service Function Chain seems like a good and equally innovative option, this solution would not be implemented without any constraints to be taken into account. The network overload could be problematic, since extra data will have to be carried within transaction packets on a regular basis. As a consequence, packets' fragmentation maybe required to ensure the overall network performance doesn't suffer a degradation. Moreover, logs' storage should also be thought about carefully, and finding the most adequate storage solution for this use case might present

another problem. A solution that may be possible is finding an external storage deposit that interacts with the SFC and bitcoin network via an API. The details of the proposed solutions should depend on the implementation and the used platforms.

Openstack Tacker

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface [32]. It can also be defined as an IaaS (Infrastructure as a Service) in terms of the services offered in the cloud computing context. Through its various components, Openstack allows global and centralized management of the different resources of a datacenter (VM instances, networking, orchestration,...etc). Launched initially by Nasa and Rackspace in 2010, the platform's success allowed it to expand gradually, adding new components for datacenter's management as new versions of the IaaS provider came through. The 2016 Openstack version (Mitaka) was the first one to include the NFV management component Tacker. Currently, Tacker is still under continuous development (version 0.7 in the time of writing of this report), with Openstack looking to provide component that is able to manage Virtual Networking Functions (VNF) efficiently either through CLI or the Openstack's web interface Horizon. In the next subsection, we take a deeper look at the networking services provided by Tacker as we explain the NFV MANO standard framework for networking components management.

NFV MANO Framework

As explained before in this report, NFV comes with improved flexibility in terms of providing networking services. Beside the high availability that could be provided through replication VNF instances inside the network, networking services can be easily scaled up and down depending on the network's load. However, it is important to define an entity that will be in charge of managing and launching these VNFs. For this purpose the European Telecommunications Standards Institute Industry Specification Group (ETSI ISG NFV) defined a framework for the management and orchestration of NFV [33].

In order to ensure efficient management of VNFs, the NFV MANO framework suggests 3 main functional blocks [34]

Virtualized Infrastructure Manager (VIM) VIM manages NFVI resources in “one domain”, as there may be multiple VIMs in an NFV architecture, each managing its respective NFV Infrastructure (NFVI) domain. NFV Infrastructure include physical as well as virtual resources (servers, storage, VMs, hypervisors...etc). The Virtual Infrastructure

Manager's is mainly charged with managing the life cycle of virtual resources in an NFVI domain. That is, it creates, maintains and tears down virtual machines from physical resources in an NFVI domain. It does also allocate physical resources to virtual machines, in addition to managing the performance and fault management of hardware, software and virtual resources.

Virtual Network Function Manager (VNFM) In the same way that VIM manages NFVI resources, VNFM is responsible for managing VNFs resources. This resources' management include the lifecycle of VNFs, their FCAP (Fault, Configuration, Accountability and Performance) and does also scale up/down the VNFs depending on the network's load.

NFV Orchestrator As the 2 previous components deal with their respective subcomponents management, the NFV orchestrator is doing global coordination of the entire network. In fact, NFVO provides linking between VIMs and NFVIs through virtual interfaces. Moreover, it creates end to end service between different VNFs. It achieves this by coordinating with the respective VNFMs so it does not need to talk to VNFs directly. For example, NFVO creates a service between the base station VNF's of one vendor and core node VNF's of another vendor. More importantly, It does the topology management of the network services instances (i.e: VNF Forwarding Graphs).

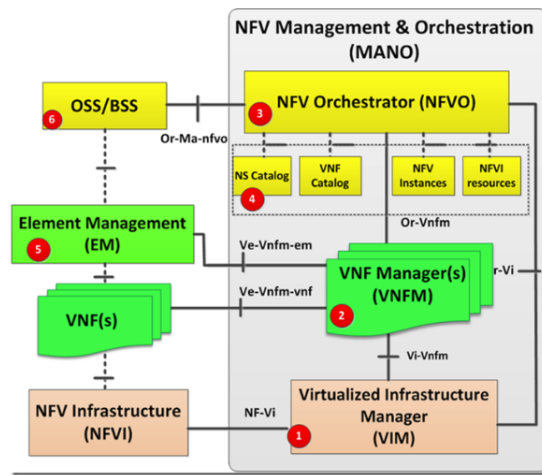


Figure 12: the NFV MANO framework architecture

Does Tacker implement NFV MANO ?

Tacker is an official OpenStack project building a Generic VNF Manager (VNFM) and a NFV Orchestrator (NFVO) to deploy and operate Network Services and Virtual Network Functions (VNFs) on an NFV infrastructure platform like OpenStack. It is based on ETSI MANO Architectural Framework and provides a functional stack to Orchestrate Network Services end-to-end using VNFs [35]. As Tacker delegates the infrastructure management to its other components, namely Heat and Keystone (as shown in the Tacker's architecture the figure below), then the Openstack NFV platform does not fully implement the NFV MANO framework.

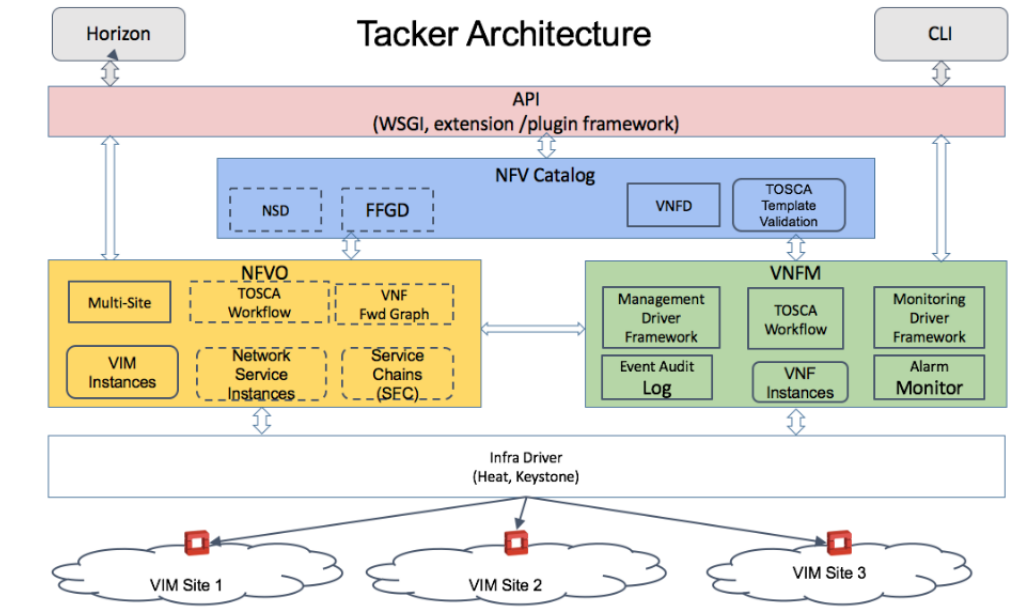


Figure 13: Openstack's Tacker architecture

Analysis & Conception

In this section, we lay the groundwork for the implementation of our solution of securing the SFC using the Blockchain technology. We do also take into account some preliminary considerations for the conception of our solution, in regard with the the different tools that we will use for this implementation.

Problem Space Considerations

As we gained the necessary knowledge about the various concepts, tools and plaforms that we could use to achieve this work, it was also paramount to actually use these tools (for the first time) and define the feasible use cases and scenarios, especially considering that the Blockchain/Bitcoin IT use cases as well as NFV platforms are still relatively new (Bitcoin core v0.14.1, Tacker v0.7). As we began with basic use cases of Openstack Tacker (devstack installation), we found different issues when we tried to create a VNFFG (Virtual Network Functions Forwarding Graph), which is basically a specification of a Service Function Chain. As previously mentionned in this report, the Openstack Tacker project has not yet reached a reliable degree of maturity that could allow us to secure the SFC components using the Blockchain (also mentionned before, 19 official bugs in 2017 as the time of writing of this report). Therefore, we decided to change some details about our initially defined problem space, in the sense that, instead of working on SFC's accountability in Tacker, we would simply work on the whole platform's (i.e. Tacker) accountability. In the next subsections, we present in more details our suggestion to secure Tacker.

Logs Immutabilization

As we've chosen to work on Tacker's accountability, we should ideally aim to secure the logs generated by Tacker's server, in order to ensure the authenticity and integrity of the information registered in thes logs. Given the Blockchain's main strength is to keep authentic records of Bitcoin transactions, our goal is to adapt the Blockchain in order to secure Tacker logs. In our attempts to find a convenient mechanism to immutabilize Tacker logs using the Blockchain, we found a research work that was already done in that regard [36]. Although it was only a proof of concept of how logs can be protected from content alteration using Blockchain, it provided us with the necessary mechanisms to secure our logs before storing them in the distributed ledger. It also gave us ideas about how to achieve the immutabilization process using the OP_RETURN field without sending too many transactions

(these details will be discussed later in the report). However, beside the fact that our work is Tacker oriented, it was also based on different technology choices as well as different design decisions. Therefore, this is purely a new (personal) implementation of the log immutabilization process provided in the research paper. In the next subsection, we explain the mechanisms of securing logs suggested in research work mentionned before.

Immutable Logs Mechanisms [36]

The secure logs are an event logging technology called immutable logs [37]. This technology implements cryptographic measures to preserve the integrity and authenticity of logs, without compromising the performance of the system. The system is based on chaining the log entries using a combination of Message Authentication Codes (MACs) and Digital Signatures (DAs). Each logger has a pair of signing keys, thus the log authenticity and non-repudiation is guaranteed. To make the logs more secure, the logging process comprises of two types of log entries: the regular log entries and the checkpoint ones. Each log entry (L_i) is chained with the previous one using a MAC cryptographic function (specifically a HMAC [38] in the implementation used). The equation below shows the form of each regular log entry

$$L_i = (LogInfo_i, h_i) \quad \text{where} \quad h_i = HMAC(K_j, (h_{i-1}|LogInfo_i)) \quad (1)$$

As L_i is the regular log entry and $LogInfo_i$ is the logging text info, each regular log entry is a concatenation of the logging info text with the entry hash. This hash is calculated as shown in the equation on the right, where we compute the HMAC cryptographic function of the previous entry hash h_{i-1} concatenated with the current logging info text $LogInfo_i$, and a random arbitrary session number K_j .

As for checkpoint entries, they are a special type of entries that are used to guarantee the authenticity and non-repudiation of the last block (j) of entries. A checkpoint Chk_j is issued every a certain number of lines or given time, depending of the logger configuration. The checkpoint entries are issued following the equation below

$$Chk_j = L_i = (LogInfo_i, K_{j-1}, E(P_{enc}, K_j), Sig_j, h_{i-1}, h_i) \quad \text{where} \quad (2)$$

$$h_i = HMAC(K_b, (h_{i-1}|K_{j-1}|LogInfo_i))$$

$$Sig_j = S(S_{sig}, (h_{i-1}|K_{j-1}|E(P_{enc}, K_j)|h_i|LogInfo_i))$$

In each checkpoint the MAC session key used to chain the last block of entries K_{j-1} is disclosed and a new one K_j , kept secret by public key encryption P_{enc} ,

is generated. Finally, a digital signature of the entry is also created with the signing key S_{sig} . Thus, any log entry manipulation or deletion is detected during the verification process.

Given that we generate a checkpoint entry (and a random session number) after an arbitrary number of lines, this will allow us to detect the exact location of the logs' content alteration. Also, the secure logs can be replicated in a central server to ensure their availability and to prevent any external attacks from changing the logs and checkpoints that has already been verified.

Components

In order to implement this log immutabilization process, we have to define the different components as well as their interactions that will immutabilize Tacker's logs. Furthermore, the immutabilization process need to be associated to another verification process that compares and validates the checkpoints. Obviously, this validation is accomplished through comparing the checkpoints issued by the logs immutabilizer with the ones stored in the Blockchain. The components diagram below sums up the our proposition for the immutabilization and validation processes. As illustrated in the diagram,

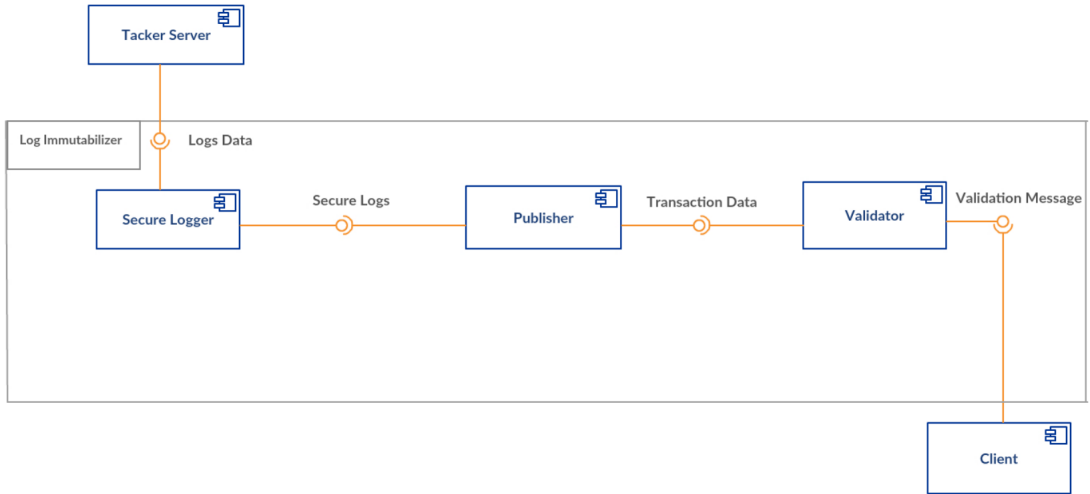


Figure 14: Logs' immutabilization and validation processes' components diagram

the Tacker server generates the logs subject to the immutabilization. Afterwards, the secure logger component receives the login data to generate new secure logs in accordance with the equations mentioned previously. The publisher component reads the secure logs and extracts checkpoints to send

them in a Bitcoin transaction and stores the transaction id. The validator receives the transaction to id in order to check the transaction's content for the OP_RETURN message (which contains the checkpoint's hash). Finally, the client component can read the validation message sent by the validator containing information about the whether the recent checkpoint has been validated or not. As we've explained the secure logging process previously through equations, we take a deeper look about the subcomponents of the publisher and validator components as shown below.

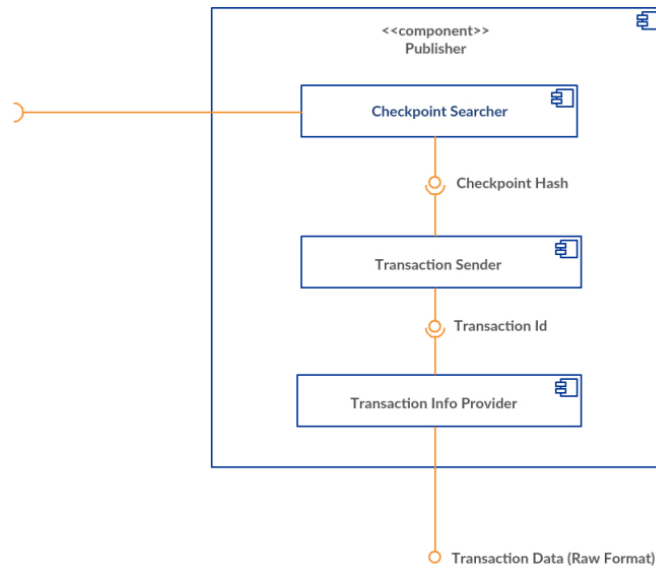


Figure 15: The publisher subcomponents

For the publisher component, a checkpoint searcher subcomponent looks for the checkpoint hashes in the secure logs, in order to provide them for the transaction sender. Afterwards, the transaction sender includes the hash in its OP_RETURN field and sends the transaction to the Bitcoin's network. Finally, the transaction info provider stores the different information related to the recent transaction that has been sent, looks for the transaction id, and then provides the transaction data in its raw (hexadecimal) format.

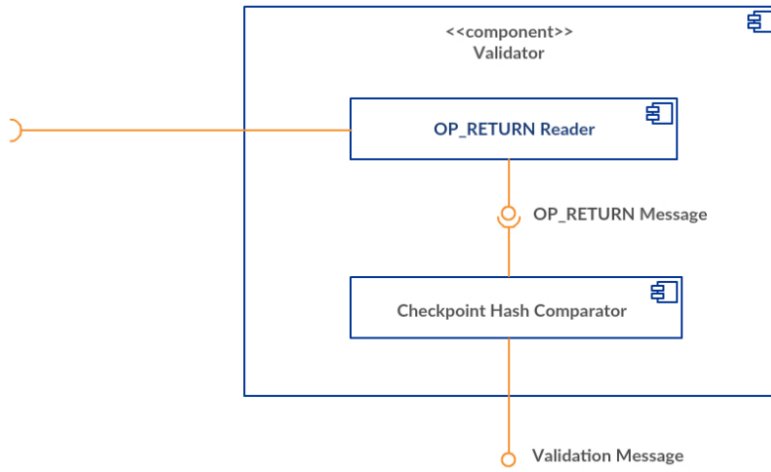


Figure 16: The validator subcomponents

As for the validator, its `OP_RETURN` reader subcomponent receives the transaction data in its raw format, checks for the message included in the `OP_RETURN` field, and then forwards this message to the checkpoint hash comparator, which compares the `OP_RETURN` message with the checkpoint hash in the log file. A validation message of the output of the comparison is sent eventually to the client. The different interactions between the components involved in the immutabilization process is illustrated in the sequence diagram below.

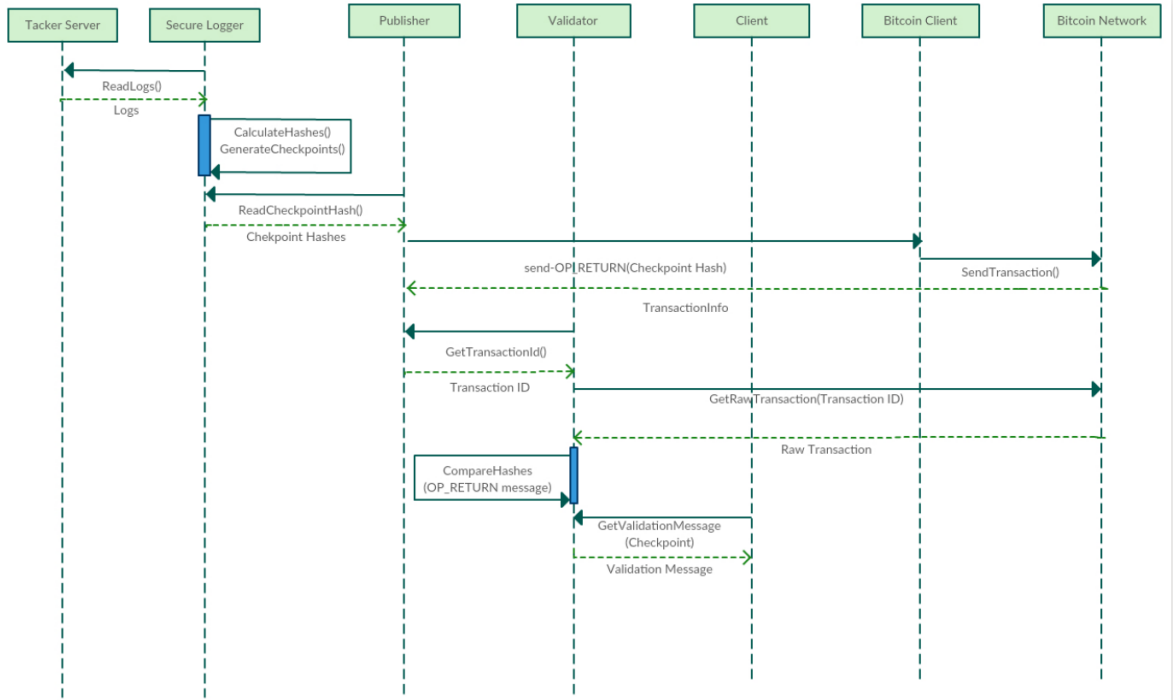


Figure 17: The sequence diagram of the immutabilization process

In addition to the interactions explained previously between the components figuring in the components diagram, we decided to add 2 new components in the sequence diagram. The addition of the bitcoin client and the bitcoin network entities in the sequence diagram was not only to show the extent of the interaction within the system, but it was also essential for acquiring an accurate view about the messages sent and received by the components, especially the publisher and the validator, which are the only system components that interact directly with the Bitcoin network.

Implementation

As we defined the various components of the logs immutabilization system and how these components should interact, we worked afterwards on implementing the logs immutabilizer. Before discussing the details of the implementation and the design decisions that we made, we ought to do a briefing of the different technologies that we used for our implementation and the motives of our choice specific of these technologies.

Technologies

1. OP_RETURN Python Library

The OP_RETURN library allows sending messages within Bitcoin's transactions. This library allows, in principle, 3 operations: send, store and retrieve, although the store and retrieve operations are not working properly.

send operation This operation allows sending a simple Bitcoin transaction containing an OP_RETURN message of 38 bytes (the library does not use the full 80 bytes of the OP_RETURN field). The command to send an OP_RETURN message is as follows

```
python send-OP_RETURN.py <receiving address> <amount>
'<message>' <0,1> (0 for the mainnet, 1 for the testnet).
```

store operation The store operation was meant to send messages with bigger length than the 80 bytes which the OP_RETURN field allows. it would also chain the transactions that contain these messages together and returns a reference for the entire message containing chain in the output. Similarly to the previous operation, this operation is issued with the command

```
python store-OP_RETURN.py <receiving address> <amount>
'<message>' <0,1>.
```

retrieve operation This operation was originally implemented in order to extract the entire message contained in several chained transactions (more than 80 bytes messages). To extract the entire message, it need the reference number returned by the store operation. The command for this operation is

```
python retrieve-OP_RETURN.py <reference number>
```

2. Bitcoin Core [39]

Bitcoin Core is an open source project that maintains and publishes the Bitcoin client software. It is a direct descendant of the original Bitcoin client created by Satoshi Nakamoto following the famous Bitcoin whitepaper. Bitcoin Core has 2 main components: a "full node" (a program that fully validates transactions and blocks [40]) software that ensures the integrity of the transactions published in the blockchain, as well as a bitcoin wallet that maintains the node's amount of BTCs. The Bitcoin Core project has a pretty much big community, especially open source programmers who contribute occasionally in the software's source code. Other programmers contribute in research work, tests and documentation.

RabbitMQ Messaging [41]

RabbitMQ is a popular message broker typically used for building integration between applications or different components of the same application using messages. RabbitMQ provides a messaging system for communication between its components. From a global perspective, there are two kinds of applications interacting with the messaging system: producers and consumers, although an application could be producer and consumer at the same time. Producers are those, who sends (publishes) messages to a broker, and consumers, who receive messages from the broker. Usually, these programs (software components) are running on different machines and RabbitMQ acts as a communication middleware between them.

Components Implementation

After defining a convenient conception for the logs immutabilization system and the technologies that will be used for it, the only phase left was to actually implement our system. As the logs immutabilizer is implemented as a proof of concept of our suggestion to improve Openstack Tacker's accountability, our implementation was more about to what extent the mechanisms that we proposed are reaching the set targets, than the details of the sources of the incoming logs data. Therefore, The components architecture proposed in the analysis & conception section would interact with simple files as sources of data.

Before starting implementing the first component, we made sure that we stored a sample of the logs generated by the Tacker server in a file, in order to pass it as input for the first component in our architecture, which is the secure logger. The secure logger, as mentioned previously, implemented the


secure logs mechanisms and relied on the HMAC function to generate the hash lines for the regular entries as well as the checkpoint entries.

```
cal/lib/python2.7/dist-packages/oslo_config/cfg.py:2770[00m, this line hash is: 30b5429ad1e28f11e51cd22de4de42a7b7b351be
cfg.py:2771[00m, this line hash is: 7e189811cbdc595587c2ccca54dd0be5df6471fa
om (pid=18738) log_opt_values /usr/local/lib/python2.7/dist-packages/oslo_config/cfg.py:2772[00m, this line hash is:

es/oslo_config/cfg.py:2774[00m, this line hash is: c2654b229f4ddb816df3716feda020af50273f59
cal/lib/python2.7/dist-packages/oslo_config/cfg.py:2775[00m, this line hash is: 826180eb1b81387e7da1066554dfce5948a6fd4d
/cfg.py:2784[00m, this line hash is: a82a3168da851e8f63620f67d5eb074df6cb3a7a
g/cfg.py:2784[00m, this line hash is: c9ae9856a252790170754a7caec9375864da9e04
g/cfg.py:2784[00m, this line hash is: 1b6cd1b150c389eebcbfe7d741256f9380de7b0e
.py:2784[00m, this line hash is: 648db3d0e547e5efe7ba8c1666fc905711d55aa1
lo_config/cfg.py:2784[00m, this line hash is: 0c57fc3f3ef821edf7f8d69d90e4e4fd848150bf
g.py:2784[00m, this line hash is: a7f022ea674d702ac1ba9515f9742c5a361a5903
nfig/cfg.py:2784[00m, this line hash is: 26257e24c3277d5d3be8c1d38c72fd9e7ff4e7ff
/cfg.py:2784[00m, this line hash is: b2fb2de720aff4b063a4db938f0c4d2b69d04635
fig/cfg.py:2784[00m, this line hash is: 10b5600835f5bce4dd8b90ea53e0c7898153cfa6
/cfg.py:2784[00m, this line hash is: 7fefa67d15afee7111ac33b9a97a2e067fb339c1
fg.py:2784[00m, this line hash is: adcc9a05bafbbdea78dc22d9f276bb6559ec0221
thon2.7/dist-packages/oslo_config/cfg.py:2784[00m, this line hash is: fcdbf45d00a93b9792f25db5c7388b370abbf53c
ig/cfg.py:2784[00m, this line hash is: 8c4b844ffeb5f5030cee891db7cb3730782e791
```

Figure 18: The line hashe concatenated at the end of each logs entry

Afterwards, another script would read the secure logs file generated by the secure logger script in order to look for the checkpoint hashes, only to store them apart in another new file. This operation would allow the publisher component to send directly those checkpoint hashes in the OP_RETURN field of a Bitcoin transaction using the python OP_RETURN library.



```
GNU nano 2.5.3 File: checkpoint_hashes.txt
fffbccdb8e8f5360400e74127e49af08f9627dd575cd2ec1cde5ffcc00a999d0ef
5f988577ce850408736cd46974fcf0c5ad8eb4db51201d9aaf80f4c70acce7d
74e5a93cdf73ebe822e4e3cd5057655fe58f3d71f713829e00adb54a711092d0
f59774b758a53762795ab9c30fa15ac35784be77b3fff6e21edb284ba9b94a5c
63cc32b18cc1765c3b515a213969d1626a45f4c6e30de282bd535995d01120d
865bdc38a432392fd8f910f49f474539111141df92a804e88f3c4345be86cd7d
7100363912f5e6db74d5f9586e60b2dc2e7bea8f3c7af099f5eb5c6ff5520cbc
560ace4c38c303b99fffc10d304fb11608c63fc75397cbf7dab82a81e3d0f90d
7e8906db32d26bf511af119b8e58e8abe3b8dcd566262315b3eed23c0ee68b24
831ba4df5ce109491e77e9ae773515d4caf8933d82318a81ca769b673343a596
0d740b7380d9f3442d5b7851721ef2b317aa6725c14c5b8156271c911ec6b939
790fb4f3771d708566ecb725dee460c8e2105f04ebb4d64adfd4d2ac7bb6baae
138cd6f7670a888c9a127719b86912002c70c542bc05689d09cb56ad0b75b109
```

Figure 19: The checkpoint hashes read from the secure logs and stored in a new file

As explained before, the store (and retrieve) operations have some implementation issues (the store operation does only send the first 32 bytes of the entire message), so we had to use the send operation to send our transactions. For automation issues that will explained in details in the next subsection, we used a bash script to send a Bitcoin transaction per hour, containing a hash read from the checkpoint hashes file in its OP_RETURN field.

♥	2017/6/19 14:58	Sent to	🔗
♥	2017/6/19 13:58	Sent to	🔗
♥	2017/6/19 13:54	Sent to	🔗
♥	2017/6/19 13:19	Sent to	🔗
♥	2017/6/19 12:58	Sent to	🔗

Figure 20: Bitcoin transactions sent once an hour at 12:58, 13:58 and 14:58

This same bash script would take the output of the send transaction operation (which contains the transaction's id) and store it in a file.

```

TxID: 6b66d9936dabeeb34b74abb6bc84e9af8f94221369910bc3ca84b3b5789320e1
TxID: 36222e8f0f7b2608260ba748713034f891cbab5163de804f7064eb482171400a
TxID: dc6ebbb077fe3c98568d2ab46496926ae7ce09465d05136c4a3277677339ce5b

```

Figure 21: The stored transaction ids

This file containin the transaction ids issued by the publisher is afterwards read by the first producer script. It reads the transaction id and then put it in the "transactions" queue. Then, the consumer script reads the transaction id from the "transactions" queue and issues the **bitcoin-cli getrawtransaction** command with the transaction id read from the queue passed to it as an argument. This allows the consumer script to obtain the full hexadecimal format of the transaction and then put it on the "op_return hashes" queue (the consumer script acting both as a consumer in the "transactions queue and producer in the "op_return hashes" queue). The validator script reads the raw format transaction from the op_return hashes queue and then looks for the prefix **"6a20"** in the message, as "6a" is the prefix for the op_return message and 20 is for the op_return's message length in bytes, which is also the length in bytes of the checkpoint hashes (32 is 20 in hexadecimal format). After the validator script reads the op_return message from the queue, it do a comparison with the checkpoint hash in the file. If the 2 strings are similar, a message of the validation of the checkpoint is put in the "validation message" queue for the client script to read. Otherwise, the validator script informs the client about the 2 checkpoints that a security breach has taken place within.

```

yassine@yassine:~/Downloads/project$ python client.py
Waiting for Messages
checkpoint number 1 has been validated

```

Figure 22: The client python script reading the validation message from the queue

Design Decisions

Although we took a simple approach in our technologies' choice as well as in our design decisions, we still faced some issues especially when it came to interacting with Bitcoin Core and the Bitcoin network. We should mention that as the Bitcoin network is growing in terms of its peers (and blocks) exponentially, there is obviously still major scalability issues existing within the network [42]. Therefore, it was unrealistic for our system to send all the hashes in transactions at once, so we've set the arbitrary frequency of sending a single transaction per hour. Moreover, we still faced problems when importing the `OP_RETURN` library directly in our python scripts, so we had to rely on a bash script executing the operation as a CLI bash command to overcome the issue (the `OP_RETURN` library should be used with great caution). Finally, before thinking about using the RabbitMQ technology, our ideal proposition was to chain transactions by manipulating their inputs and outputs, so that eventually we'll need only the first transaction to read the entire hashes stored in the chain. Unfortunately, manipulating Bitcoin's fields can be hard, complicated and time consuming (as proven here [43]), so we had to take the simple approach of using a message broker that would move the transaction ids between the components and would allow them to extract the hashes stored within as simply as possible.

Conclusion

References

- [1] J. Halpern & C. Pignataro, *Service Function Chaining (SFC) Architecture*, RFC7665.
- [2] D. Migault, Ed. Ericsson, T. Reddy & C. Pignataro, *SFC environment Security Requirements*.
- [3] <https://bugs.launchpad.net/tacker>
- [4] <https://www.ascribe.io/>
- [5] <https://blocksign.com/>
- [6] Guardtime secures over a million Estonian healthcare records on the blockchain, <http://www.ibtimes.co.uk/guardtime-secures-over-million-estonian-healthcare-records-blockchain-1547367>
- [7] Problem Statement for Service Function Chaining, <https://www.rfc-editor.org/rfc/pdf/rfc7498.txt.pdf>.
- [8] The potential for blockchain to transform how organizations produce and capture value is very real ,but so are the challenges to its broad implementation. <http://sloanreview.mit.edu/article/seeing-beyond-the-blockchain-hype/>
- [9] Bitcoin: A Peer-to-Peer Electronic Cash System, <https://bitcoin.org/bitcoin.pdf>
- [10] <https://blockchain.info/fr/charts/n-transactions-per-block>
- [11] Bitcoin Wiki: Protocol Documentation, https://en.bitcoin.it/wiki/Protocol_documentation#Merkle_Trees
- [12] Bitcoin Developer Guide, <https://bitcoin.org/en/developer-guide#block-chain>

- [13] <https://www.cryptocompare.com/>
- [14] <https://en.bitcoin.it/wiki/Nonce>
- [15] <https://www.bitcoinmining.com/>
- [16] <https://www.ethereum.org/>
- [17] <https://trends.google.com/trends/explore?q=blockchain>
- [18] Blockchain Technology for Recordkeeping, Help or Hype ?
(Appendix C: Blockchain companies),
https://www.researchgate.net/profile/Victoria_Lemieux/publication/309414363_Blockchain_for_Recordkeeping_Help_or_Hype/links/580f539408ae009606bb62f6.pdf
- [19] Blockstack: A Global Naming and Storage System Secured by Blockchains,
<https://blockstack.org/blockstack.pdf>
- [20] Bitcoin: A Peer-to-Peer Electronic Cash System, <https://bitcoin.org/bitcoin.pdf>
- [21] What is proof of existence ?, <https://proofofexistence.com/about>
- [22] Hyperledger Whitepaper, <http://www.the-blockchain.com/docs/Hyperledger%20Whitepaper.pdf>
- [23] <https://www.hyperledger.org/community/projects>
- [24] Architecture of the Hyperledger Blockchain Fabric,
https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf
- [25] An empirical study of Namecoin and lessons for decentralized namespace design,
http://www.econinfosec.org/archive/weis2015/papers/WEIS_2015_kalodner.pdf
- [26] Explanation of what an OP_RETURN transaction looks like,
<https://bitcoin.stackexchange.com/questions/29554/explanation-of-what-an-op-return-transaction-looks-like>
- [27] OP_RETURN 40 to 80 bytes, http://www.talkcrypto.org/blog/2016/12/30/op_return-40-to-80-bytes/
- [28] Bitcoin Wiki: Script, <https://en.bitcoin.it/wiki/Script>

- [29] Bitcoin Wiki: Testnet, <https://en.bitcoin.it/wiki/Testnet>
- [30] TP's Testnet Faucet <http://tpfaucet.appspot.com/>
- [31] Neutron API for Service Chaining, <https://wiki.openstack.org/wiki/Neutron/APIForServiceChaining>
- [32] Openstack Official Website, <https://www.openstack.org/software/>
- [33] An introduction to NFV and the OpenStack Tacker project, <https://blog.imaginea.com/an-introduction-to-nfv-and-the-openstack-tacker-project/>
- [34] A Beginner's Guide to NFV Management & Orchestration (MANO), <http://www.telecomlighthouse.com/a-beginners-guide-to-nfv-management-orchestration-mano/>
- [35] Openstack Tacker Wiki, <https://wiki.openstack.org/wiki/Tacker>
- [36] Distributed immutabilization of secure logs, https://www.scytl.com/wp-content/uploads/2017/01/Distributed-Immutabilization-of-Secure-Logs_Scytl.pdf
- [37] A.O. Cornet and J.M.B. Bosch. Method and system of generating immutable audit logs, January 15 2009. US Patent App. 12/096,048.
- [38] National Institute of Standards and Technology. FIPS 198-1, The Keyed-Hash Message Authentication Code (HMAC), Federal Information Processing Standard (FIPS), Publication. Technical report, U.S. Department Of Commerce, July 2008.
- [39] L'actualité de Bitcoin en français, <https://bitcoin.fr/bitcoin-core/>
- [40] Bitcoin Core Official Page, <https://bitcoin.org/en/bitcoin-core/>
- [41] Introduction to RabbitMQ, <http://www.baeldung.com/rabbitmq>
- [42] Bitcoin Scalability Problem Explained, <https://cointelegraph.com/explained/bitcoin-scaling-problem-explained>
- [43] Bitcoins the hard way: Using the raw Bitcoin protocol, <http://www.righto.com/2014/02/bitcoins-hard-way-using-raw-bitcoin.html>