

Enhancing SDN Access Control with Private Ethereum Blockchain

MSc Research Project
Cybersecurity

Mariusz Graczyk
Student ID: x20197446

School of Computing
National College of Ireland

Supervisor: Mr. Ross Spelman

National College of Ireland
MSc Project Submission Sheet
School of Computing



Student Name: Mariusz Graczyk.....
Student ID: x20197446.....
Programme: MSCCYBETOPYear: 2024
Module: Research Project
Supervisor: Mr. Ross Spelman
Submission Due Date: 12 August 2024
Project Title: Enhancing SDN Access Control with Private Ethereum Blockchain...
Word Count: 7811 Page Count 20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: *Mariusz Graczyk*

Date: 16 September 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	<input type="checkbox"/>
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhancing SDN Access Control with Private Ethereum Blockchain

Mariusz Graczyk
x20197446

Abstract

Software Defined Networks (SDN) have been implemented in various environments requiring agility, rapid and automated response to changing network circumstances and centralised management control of all network components. This is achieved by decoupling the data plane from the control plane in the SDN paradigm. Cloud Computing and Internet of Things are the most common environments where SDNs can handle large-scale network traffic loads efficiently and dynamically. However, the central management control of the entire network has also its drawbacks. Blockchain (BC) technologies come with numerous security-enhancing benefits and BCs offer decentralised design which could effectively complement centrally controlled SDNs. This paper highlights various aspects of combining BC with SDN to address inherent security concerns related to SDNs. BCs address all three aspects of the CIA Triad and this paper discusses their main benefits in the context of SDNs, which are decentralised design, data immutability and integrity, enhanced authentication and non-repudiation. Ethereum BC networks and the Proof of Authority (PoA) consensus mechanism are of particular interest in this paper. The PoA consensus is best suited for private Ethereum BC implementations.

1 Introduction

This research project investigates the security benefits of implementing private Ethereum BC technologies to enhance the access control of data-plane and end-user devices operating in OpenFlow SDN environments. While SDN architectures offer numerous benefits, such as simplified management, operational network efficiency, enhanced flexibility, and reduced reliance on vendor-specific solutions, they also raise significant concerns, such as the controller being a single point of failure, Denial of Service (DoS) attacks, device spoofing attacks or Man-In-The-Middle (MITM) attacks.

1.1 Research Question

“What security benefits are gained from integrating SDN Networks with private Ethereum Blockchain?”

1.2 Objectives

The project analyses the benefits and implications of integrating private PoA¹ Ethereum BCs with SDN networks, focusing on the network infrastructure and end-user access control. The objectives will be achieved by investigating four scenarios covering Transport Layer Security

¹ <https://ethereum.org/en/developers/docs/consensus-mechanisms/poa/>

(TLS) encryption, controller-generated certificates used for second-factor authentication, failed login events stored in the BC as immutable logs, and a Metamask-based authentication mechanism for end-user devices.

2 Related Work

SDN networks are still an emerging network paradigm, but have attracted attention of research communities and industries in recent years, due to their network management and control. SDN networks decouple the control plane from the data plane, as a result centralising network management and allowing network programmability [1]. SDN-based solutions have gained prominence especially in Cloud Computing environments and in heterogeneous IoT infrastructures, where SDN networks are capable of processing large amounts of data efficiently and rapidly respond to changing circumstances. However, the centralized aspect of SDN networks exposes them to several types of potential attacks. Thus, SDNs could be complemented by BC, another emerging technology, in a combined solution for enhanced security. BC provides “a decentralized ledger for transactions and data security, preventing unauthorized access and tampering” [2]. The participants in BC transactions remain in control of their data and reliance on third-parties is eliminated [3]. Centralised SDNs are complemented by distributed peer-to-peer BC nodes, which various researchers highlighted as a significant advantage enhancing network security, privacy and efficiency [3].

2.1 SDN data-plane vulnerabilities and security solutions

SDNs simplify network administration and provide agility and flexibility that facilitates rapid deployment of network updates, services and applications [4]. These characteristics of SDNs are due to a global view and centralised control over all network components, such as the Data/Forwarding Plane, the Southbound Interface, the Control Plane, the Northbound Interface and the Application Plane. Goud and Gidituri [4] list the main SDN security vulnerabilities and attack types, as well as the SDN components which they relate to. Data plane devices could be vulnerable to DoS/DDoS attacks, malicious flow injections/tampering, MITM attacks, TCAM buffer attacks, or TLS vulnerabilities. Agborubere and Sanchez-Velazquez [5] emphasise the fact that TLS is not a default option in the OpenFlow standard. Nevertheless, it is a pre-requisite to ensure the Southbound API and OpenFlow messages are encrypted. Gupta et al. [1] assert that data-plane devices are prime targets for attackers. They mention documented exploits, such as the NSA’s core infrastructure backdoors, or the CIA’s exploitation of Cisco routers. They also list some challenges in securing the SDN data plane, e.g. software switches like Open vSwitch being more vulnerable to attacks compared to physical SDN switches. Ohri and Neogi [6] list several solutions for securing the SDN data plane. The most common are *FortNoX*, *FlowGuard*, and *VeriFlow*. However, they assert that these solutions were designed only for specific controllers and have not been implemented in real-world environments due to performance issues, e.g. inadequate performance in multi-controller setup in the case of *VeriFlow*. Jimenez et al. [18] assert that many proposed SDN security implementations do not succeed “because they cannot be economically or technically leveraged.” The aforementioned solutions are traditional approaches to securing SDN data planes, which are contrasted by Abdi et al. [7] with new approaches to SDN security. They

assert that newer approaches to SDN security, such as Artificial Intelligence (AI) and Moving Target Defence (MTD), provide more proactive and adaptive defence strategies compared to traditional ones through enhanced detection and mitigation of sophisticated attacks.

2.2 Blockchain Technologies

The BC technology is defined by Alharbi [8] as a distributed ledger that stores a list of transactions and events forming a sequence of blocks that are managed by a cluster of computers instead of a single entity. One of the main benefits of BC is its de-centralized architecture. BC networks are categorised as a public BC (open to everybody), a private or permissioned BC (protected with restricted access) and a consortium BC (with multiple organisations sharing it). Many different BC protocols have been implemented as frameworks governing the operation of BC networks. The most well-known ones are Bitcoin and Ethereum. The latter one facilitates a decentralized platform for smart contracts and decentralized applications (dApps) based on Web3, which offers end-users more control and ownership of their data compared to well-established Web2². Ethereum recently transitioned from the Proof of Work (PoW) to the Proof of Stake (PoS) consensus mechanism with the introduction of Ethereum 2.0, which resulted in significantly lower energy consumption. According to Fahim et al. [19], the third consensus mechanism – PoA – is even better suited for private BC implementations. They assert that, compared to PoS and PoW, it ensures better transaction handling efficiency, lower energy and computation requirements, and it provides effective defence against 51% network attacks.

2.3 Integrated BC-SDN solutions for enhanced security

The below table presents selected studies focusing on the security of integrated BC-SDN solutions.

Table 1: Advantages and limitations of selected BC-SDN solutions

Study	Advantages	Limitations
Meng et al. [9]	The study introduces BSDNFilter - an IDS-based security mechanism building trust-based filtration through traffic fusion and aggregation to combat malicious traffic. The solution offers enhanced traffic management and decentralised trust mechanisms.	Some issues might arise, such as performance concerns, solution complexity, scalability issues and integration or operational costs. CPU workloads also proved to be a concern.
Latah and Kalkan [11]	DPSEc offers security enhancement for SDN data-plane devices through the use of BC. It addresses specific OpenFlow and DoS vulnerabilities.	Challenges relate to solution performance, scalability and implementation complexity
Derhab et al. [12]	This innovative solution offers multi-controller setup integrated with BC, which decentralises network management and mitigates the risk of a single point of failure. A reputation mechanism rates controllers	It only covers the security of east-west interfaces. The proposed solution introduces significant complexity which might cause integration issues with existing SDNs. Achieving synchronized multiple controllers in

² <https://ethereum.org/en/developers/docs/web2-vs-web3/>

	based on their actions, which enhances the security of flow updates	a large-scale SDN network could cause scalability concerns and would be resource-intensive
Rahman et al. [2]	DistB-SDCloud – solution enhancing security of IIoT applications in cloud computing through leveraging BC and SDN, which helps in efficient management of cloud resources. The paper is a thorough solution evaluation using specific performance metrics.	The combination of cloud computing, BC and SDN introduces significant complexity and it may lead to compatibility or interoperability issues. There might be scalability issues in large-scale IIoT deployments. As the BC grows, it may cause performance overhead for cloud computing.
Steichen et al. [13]	ChainGuard is a firewall solution protecting BC applications. It filters and manages network traffic to BC nodes. ChainGuard adapts to changing network conditions – thus, providing a more responsive solution compared to traditional firewalls. OpenFlow integration ensures more granular control of network traffic.	Lack of comprehensive practical evaluation and tests performed in production environments. ChainGuard might require increased operational costs as both BC and SDN require substantial computational power. It might introduce performance overhead when large-scale traffic needs to be filtered and processed.
Houda et al. [14]	ChainSecure is a scalable solution integrating SDN with BC which provides proactive security measures to protect BC against network-related threats. ChainSecure optimises routing of Blockchain transactions, thus, reducing latency.	This solution is tested only using Mininet virtual environment rather than real physical devices. It protects BC but relies on SDN which itself may be vulnerable to various attacks. A central controller constitutes a single point of failure.
Sharma et al. [15]	DistBlockNet provides a promising approach to improving the management and security of large-scale IoT networks by combining SDN with BC. It offers a decentralized multi-controller architecture integrated with decentralized BC nodes.	Although it was tested with 6 controllers and 6000 software nodes, it still raises performance overhead and resource consumption concerns when it is implemented in real-life production environments. Moreover, compatibility issues may arise when integrating heterogeneous IoT devices with SDN and BC.
Hu et al. [16]	The study introduces an innovative solution integrating edge computing, SDNs and BC. It recommends using a BC-as-a-Service (BaaS) provider to integrate the BC aspect into the solution, which might reduce the total cost of ownership. The control is distributed among the edge devices and SDN switches are coupled with BC agents.	The integration of BaaS into the solution might lead to regulatory issues in heavily regulated industries. Although the control is transferred to the network edge, this might create new attack surfaces when the targets are closer to the network edge. The combination of edge computing and BC could put a strain on resource-constrained IoT devices.
Faizullah et al. [17]	The paper proposes an IoT cloud solution based on OpenFlow SDN and permissioned BC. The permissioned BC is contrasted with public BC and experiments indicate better efficiency of permissioned BC for handling IoT devices at a large scale.	Although the experimental results are promising, this solution has not been tested in a real-life production environment. Despite permissioned BCs being decentralised, a limited number of miner nodes controlling the BC might lead to scalability issues. Even when permissioned BCs are less resource intensive than public BCs, they still require substantial computational resources, which may put a strain on IoT devices.

The most prominent BC benefits highlighted in the literature include the potential for enhanced data security, integrity and immutability, as well as anonymity, privacy and transaction transparency. BC technologies have increasingly started to be integrated with SDNs in various design contexts. Some of them emphasise the benefits of the BC-SDN integration for large-scale IoT networks [13][16][17]. The usefulness of BC-SDN integration offers great potential for cloud computing technologies [13][17], or when such BC-SDN integration happens closer to the network edge in the context of edge computing and taking advantage of the BaaS service [16]. Solution scalability and reduced transaction latency are highlighted in [14][17], or multi-controller architectures [12][15] are emphasised by some other studies. Better transaction efficiency and increased data security is discussed in the context of private, permissioned or consortium BC networks [16][17]. Protection against malicious flow injections [9][12] or DoS attacks [11] are also important mitigation measures discussed. Not only the SDN planes, but also the BC nodes need to be protected against attacks, which are offered as a BC firewall solution by [13]. Finally, end-user access control and authentication is the focus of [10], which incorporates a digital wallet into the end-user authentication process. The idea of digital wallets was integrated into this BC-SDN project.

Many of the BC-SDN benefits outlined in the literature laid foundations for this project, which also implemented an integrated BC-SDN solution focused on data-plane access control. Data confidentiality and integrity were ensured by implementing TLS encryption in the south-bound interface, or by implementing HTTPS POST requests. Data Availability was facilitated through the decentralised multi-node BC topology. Any data or logs saved in the BC by the SCs cannot be reverted, deleted or tampered with. This immutable characteristic of BC features prominently in many of the above studies. Moreover, the Metamask digital wallet, based on the Private Key Infrastructure (PKI), combined with the SCs and SDNs offers an effective and efficient authentication mechanism, which fulfils both the confidentiality and non-repudiation requirements. The PoA consensus was identified as most suitable for a private BC implementation, which is reflected in the literature [17][18]. Finally, implementation complexity is a recurring limitation of many of the studies. Thus, this proposed BC-SDN implementation is a light-weight solution capable of reacting automatically to certain security events.

3 Research Methodology

In order to demonstrate how the BC could enhance the security of SDNs, many technical components had to be implemented. The implementation of these components required achieving mostly binary outcomes. However, the final case-study regarding the end-user authentication mechanism involved gathering also continuous data that was analysed and compared to the study conducted by Petcu et al. [10].

3.1 Technical stages and testing performed

An Ubuntu Virtual Machine (VM) running in VMware and having network connectivity to the host operating system (OS) was created. The network connectivity to the host OS subnet was facilitated via a virtual adapter operating in the bridged mode. The Ryu package and the Mininet emulator were installed in the Ubuntu VM. A Ganache node emulating the BC node

was deployed in the host OS. The first iteration of the *SDNEther* app and the *SDNEtherMonitor* SC was created to confirm that a simple transaction could be performed. The SC was deployed to Ganache via Remix IDE.

i. First-Stage Implementation

Open vSwitch (OvS) was installed on a Raspberry Pi 4 device. A simple Mininet topology was deployed to confirm successful connectivity to the Ryu controller. The SC was updated to be able to register IP addresses and host names. When Mininet hosts were launched and connected to the controller, their details were also registered in the SC. Moreover, a Python script was created for Mininet that deployed an OvS switch, several hosts and an interface in the Ubuntu VM. Each Mininet host was deployed as an SSH server. The second iteration of the *SDNEther* app and the *SDNEtherMonitor* SC was created, i.e.: when a new switch or host was connected to the controller, the app was capable at that stage of generating a self-signed second-factor authentication (SFA) certificate and transferring it to the device via SFTP and to the SC via the Web3 connection and the Application Binary Interface (ABI) interface. The SC was capable of saving and storing the SFA certificates of devices. The port-handler method was updated in the app to ensure that whenever a device re-connected to the OvS switch port, the app automatically fetched the certificate from the device and sent it to the SC, which in turn determined if the certificates are identical. Based on this test, network access was either granted or denied. TLS self-signed certificates were generated for the OVS switches and the controller to enforce TLS encryption of OpenFlow messages. The implementation of the TLS certificates was tested with Wireshark. A Bash script was created for OvS running on the Raspberry Pi 4 device and the corresponding REST endpoint was implemented in the app to handle HTTPS POST requests sent to the app by the Bash script when three consecutive failed login events occurred. The third iteration of the app and the SC were created at that stage to facilitate saving logs generated on the OvS switch and sent by the app to the SC.

ii. Second-Stage Implementation

A Reverse *NGINX* proxy was deployed in the Ubuntu VM and on the Lab PC2 running the BC. The proxy was to listen on ports 443 (Ubuntu VM) and 8743 (PC2) and transfer incoming requests to 127.0.0.1:8080 or 127.0.0.1:8501 respectively. A test simulating three consecutive failed login attempts was performed and a successful log registration in the SC was the desired outcome. A multi-node PoA BC network was deployed in the Lab PC2 running Ubuntu OS. The *Geth* and *Puppeth* binaries were installed and the BC network consisting of three peer-nodes and a bootnode was successfully launched. A Metamask digital wallet was created and the JSON file for the miner node account was imported into this wallet. A transaction from the Metamask local account to the imported shared account was successfully performed. The *PCmonitor* SC was deployed in the BC for registering MAC addresses and associated end-user Metamask accounts. The new *PCmonitor* SC was tested successfully by performing a transaction via Remix IDE in order to register end-user MAC addresses and Metamask accounts in the SC. A Python script for monitoring the shared account was created and a new REST endpoint was added to the app. After the Python monitoring script was launched on the PC running the BC nodes, a test was carried out while

an end-user PC with the pre-installed Metamask wallet was connected to the Raspberry Pi's OvS switch. At that stage the PC had to have only limited network access to the BC node. An ICMP (ping) probe test was suitable for confirming expected limited network connectivity. A Metamask transaction with the destination of the imported shared account was performed. The Python script, which monitored the shared account for any received transactions, detected a pending transaction to this account and queried the *PCmonitor* SC for the MAC address associated with the sender's account and subsequently sent the transaction details, including the MAC address, in a JSON format via an HTTPS POST request to the *SDNEther* app. On receipt of the incoming POST request, the app in turn checked the *mac_to_datapath* dictionary for the switch port where the given MAC address was connected to. Finally, the app pushed the unblocking flows for the IP address of the end-user's PC to the relevant OvS switch. At that stage ICMP probe tests to private local IP addresses and to public external IP addresses was successful. These ping tests confirmed that the end-user PC was granted full network access.

3.2 Desired Outcomes

The desired outcomes for the expected goals were mostly binary results, i.e. was the log registered in the SC or not, or was the end-user PC granted only limited network access or full access. These outcomes could be verified by executing ping probe tests, or by sniffing the network traffic for selected hosts with Wireshark to confirm TLC encryption, or by checking the SC with Remix IDE. Another test was performed to prove that the SFA certificate authentication was successfully implemented by simulating a Mininet host getting disconnected from the network, then altering its SFA certificate, and reconnecting the host and observing if the device network access was granted or blocked. The outcome of the use-case regarding consecutive failed login attempts was verified by simulating three consecutive failed login attempts and checking if a relevant log was saved in the SC for the device. This was checked with Remix IDE.

However, the expected outcomes of the end-user PC authentication process required the evaluation of both binary and continuous data, i.e. how fast and efficient the proposed BC authentication mechanism was. Furthermore, authentication times were compared with the work of Petcu et al. [10] who proposed a BC-based authentication mechanism involving Metamask wallets as well. Once the authentication mechanism in this project was implemented, quantitative raw data, including BC transaction hashes and Unix-format timestamps, was collected at specific stages of the authentication process and analysed with the Python-based NumPy and Matplotlib libraries³⁴. The timestamps were captured at defined stages. These stages were: when the end-user clicked the final 'Confirm' button, when the Python monitoring script detected a received transaction by the shared account, when the *SDNEther* app received an HTTPS POST request from the Python monitoring script, and finally, when the unblocking flows were pushed to the authenticating PC.

³ <https://www.programiz.com/python-programming/numpy/statistical-functions>

⁴ <https://matplotlib.org/stable/tutorials/index.html>

4 Design Specification and Requirements

The implementation of the project was dependent on the app's and controller's interactions with the data plane devices and the BC. Both virtual and physical switches and hosts, deployed in the test setup, had to be able to register with the controller, which should install initial flows in the OvS and register device details and failed login logs in the BC. End-user PC authentication via a digital wallet and the BC were also to be implemented.

The diagram in Figure 1 presents a high-level design of the test lab setup.

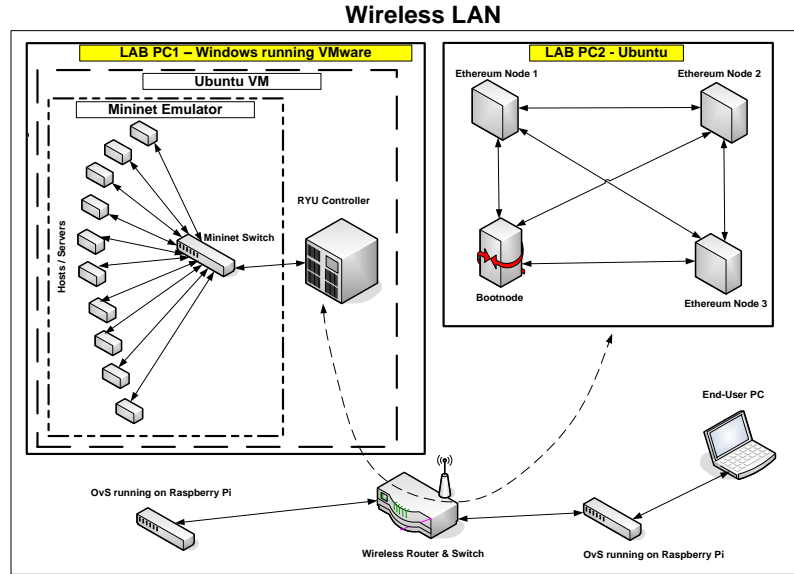


Figure1. Test lab environment consisting of two Lab PCs, wireless router with Ethernet switch ports, two Raspberry Pi 4 devices and an end-user laptop

The main design requirements for this test lab system are:

- A VM hosted in VMware would accommodate the RYU Controller and the Mininet emulator running concurrently. To validate the functionality of the test solution on physical switches, OvS installed on Raspberry Pi 4 devices are to be used.
- The controller app should interact with a private PoA Ethereum BC network consisting of several nodes. It is essential to implement a private Ethereum BC because the access to the BC should not be open to the public.
- The *SDNEther* app should be the 'brains' of the entire system, interacting dynamically with OvS, endpoint devices and with the BC to react to specific security events, querying the BC and enforcing access control as required. This access control would be based on feedback from the BC. Because the Ryu controller is based on Python, the application executing expected outcomes on the controller ought to be also developed in Python.
- The end-to-end communication involving OvS switches, Mininet hosts, the controller and the BC, must enforce TLS encryption to avoid any OpenFlow messages being intercepted or potential MITM attacks. If any component is not capable of handling

SSL communication, a reverse proxy needs to be implemented to enforce the end-to-end encryption.⁵

- e) The SC deployed in the BC needs to be able to register required details of OvS switches or Mininet hosts in the BC and it should provide feedback on already registered devices. The BC would be handling the validation of SFA certificates for access control when queried by the app.
- f) A private BC network utilising the PoA consensus mechanism should be implemented. Unauthorised nodes should be prevented from joining the BC network.⁶
- g) The end-user PC, Mininet hosts and OvS switches should be authenticated by the BC. However, the end-user PC should be using Metamask for authentication in the BC, whereas Mininet servers and OvS switches should use SFA certificates.

5 Implementation

The test lab for this project includes multiple components. It uses VMware Workstation hosting the Ubuntu VM, on which the Mininet network emulator and the Ryu controller were installed. To simulate initially a private Ethereum BC node, the Ganache node with the graphical user interface was installed on the host computer and the connectivity between the Ubuntu VM and Ganache was established via a network adapter in the NAT mode. Subsequently, a PoA Ethereum network was to be deployed on another PC running Ubuntu. To test the implementation on physical devices, software-based OvS was installed on two Raspberry Pi 4 devices and another network adapter in the bridge mode was created in VMware, so that the Ubuntu VM could share the same subnet with the OvS switches. The Mininet switch and hosts were deployed with a Python script creating the switch and servers, and pointing these devices towards the controller, since Mininet and Ryu run concurrently in the same VM. The script should also create a Network Address Translation (NAT) interface on the Ubuntu VM, allowing it to establish SSH connectivity with each Mininet server. A detailed step-by-step outline of how the test lab set-up was created is included in the configuration manual.

5.1 Deployment of Ganache node and PoA Ethereum nodes

To achieve interactions as soon as possible between the developed app and the BC, the Ganache server node was initially deployed in the Lab PC1 (see Figure 1) sharing the same network subnet with the Ubuntu VM. However, in the final stage it was replaced with a proper multi-node PoA Ethereum network. The Clique⁶ consensus engine was implemented via Puppeth deployment manager (included in the Geth package⁷), in which one node acts as a miner and transaction signer and the bootnode facilitates communication between all the three nodes. In a private PoA BC network the crypto-currency has only a nominal meaning as it cannot be used on the Ethereum Mainnet network, which suited the project purpose. Each

⁵ <https://blog.yeetpc.com/how-to-set-up-an-nginx-reverse-proxy-with-ssl-on-ubuntu-server-20-04-lts/>

⁶ <https://github.com/ConsenSys-Academy/eth-poa-tutorial?tab=readme-ov-file#configuring-clique-via-puppeth>

⁷ <https://geth.ethereum.org/docs/getting-started/installing-geth>

node was launched with `--nodiscover` flag to prevent any unauthorised nodes from discovering the current nodes.

5.2 Deployment of Smart Contracts

Two different SCs were created in the project – *SDNEtherMonitor* and *PCmonitor*. These two smart contracts were created with the Solidity programming language and deployed via Remix IDE to the BC. The former SC includes functions that register devices in the BC, validate SFA certificates, update failed login logs, confirm prior device registration and list registered devices' details. The Remix IDE was used to deploy the SCs to the Ganache node initially, and subsequently via Metamask, to one of the private BC nodes.

The *PCmonitor* SC plays a different role. It stores the Metamask accounts of authorised end-users and the corresponding MAC addresses of the end-user PCs on which their Metamask wallets were created. An additional script written in Python was deployed to monitor the BC for any transactions sent to the shared account. When such transactions occur, the script queries the *PCmonitor* SC for the MAC address associated with the sender's account, and subsequently it sends an HTTPS POST request to the *SDNEther* app. This request sends a JSON including the transaction details and the MAC address associated with the sender's Metamask account.

5.3 SDNEther App

SDNEther, a light-weight Python app developed for this project, plays a central role in this solution as it handles devices registrations in the BC and pushes the required flows to the switches. It also processes HTTPS POST requests from the OvS switches and from the Python script monitoring the BC. The following functionality was implemented in the app:

- a) Three methods in the app play an important role to ensure the app could respond to certain events, such as a device connecting to the controller, a switch port changing its status, or an IP packet handled by the app. These methods are respectively:

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_handler(self, ev):
    [ ... ]

@set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
def port_handler(self, ev):
    [ ... ]

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_handler(self, ev):
    [ ... ]
```

The decorators `@set_ev_cls(ofp_event.[...])` make these methods react to specific network events in a dynamic way.

- b) Upon initial host connection to the OvS switch, *SDNEther* detects such a connection attempt and generates a self-signed certificate for the connecting device and sends the certificate to it. The app then registers the device's name, IP address and the generated SFA certificate in the BC by interacting with the SC. Paramiko, a library imported into the app, is used by *SDNEther* for initiating SFTP connections to the hosts to transfer the certificates to them. The application communicates with the SC via Web3 - a Python library that interacts with the SC via the ABI. The *switch_handler* method handles initial device registration and subsequent access validation.
- c) The *port_handler* method monitors changes to the status of the links connected to the switches. If the status of the link changes or a device reconnects to the switch, *SDNEther* uses SFTP to fetch the previously deployed certificate from the device and checks this certificate against the certificate copy for that device stored in the SC. If both certificates are identical, the re-connecting device passes the authentication and is granted network access. Otherwise, the app blocks this device by pushing device-specific blocking flows to the switch.
- d) Each switch runs a Bash script that monitors */var/log/auth.log* file for any three consecutive failed login attempts. In the event of such failed attempts, the switch triggers an HTTPS POST request to the app and sends a log message to it. This request is received by the NGINX reverse proxy on port 443 and forwarded to the app on the URL: *http://localhost:8080*. This message includes the device name, its IP address, SFA certificate and the source IP address of the device making these login attempts. The app in turn passes the log message to the SC, which appends this log event to the relevant device's log.
- e) The methods *update_switch_log* and *handle_transaction* handle incoming *POST* requests addressed to */apps/{log}* or */transactions* endpoints respectively. These endpoint handlers process log requests sent by OvS switches or by the BC monitoring script when end-user device authentication occurs. Regarding the latter method, once the end-user initiates the authentication transaction and it is detected by the Python monitoring script, the script fetches the MAC address of the sender's Metamask account from *PCmonitor* SC and sends an HTTPS POST request to the app. Upon receipt of the request, the app needs to determine which switch port this incoming request relates to by checking the mapping dictionary *mac_to_datapath*. When the datapath is obtained, the relevant unblocking flows are sent to the switch and the end-user is granted network access.

5.4 Mininet deployment script

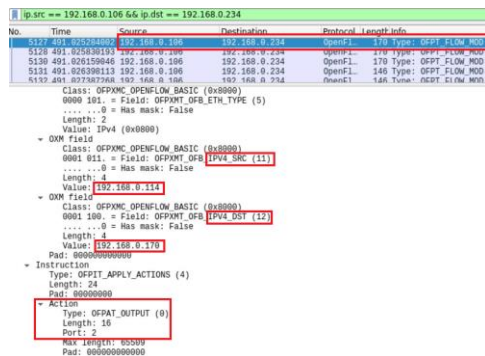
The topology deployment script is a custom Python script deploying a selected number of switches and hosts acting as SSH servers to the Mininet emulator environment.⁸ The deployment script creates an additional NAT interface in the Ubuntu VM. This interface is available to all the created hosts and to Ubuntu VM itself, which allows access to each host from the Ubuntu VM. The script also creates an SSH server on each host, which enables the app to transfer certificates to each Mininet host via SFTP sessions on port 22. Furthermore, the script also points the Mininet OvS switch to the path of its SSL certificate, private key and the CA certificate⁹. As a result, the Ryu controller is able to establish encrypted TLS communications with the Mininet-generated switch.

6 Evaluation

Several use-cases were created to investigate the potential of the BC to enhance the access-control security of the OpenFlow SDN networks. The main characteristics of the BC that these use-cases benefit from security wise are distributed architecture of the BC, strong authentication based on PKI and data immutability and non-repudiation of transactions. Another benefit that the BC could offer is anonymity, although it was not explored in this project.

6.1 Case-study 1: TLS encryption

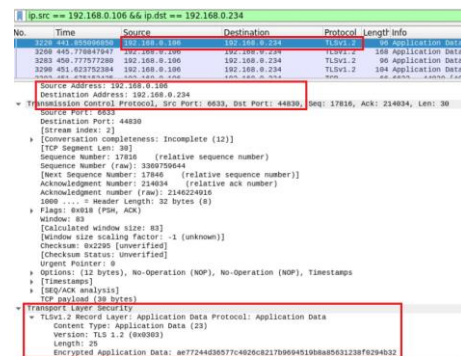
The OpenFlow protocol does not enforce TLS encryption by default, which might create an opportunity for intercepting data in transit between the OvS switches and the controller if such a configuration is left at its default. At the very least, a bad actor could sniff and intercept unencrypted OpenFlow network packets and obtain a detailed view of communications between the SDN switches and the controller.



No.	Time	Source	Destination	Protocol	Length	Info
5127	4.91	192.168.0.106	192.168.0.234	OpenFlow	510	Type: OFPT_FLOW_MOD
5128	4.91	192.168.0.106	192.168.0.234	OpenFlow	270	Type: OFPT_FLOW_MOD
5129	4.91	192.168.0.106	192.168.0.234	OpenFlow	170	Type: OFPT_FLOW_MOD
5131	4.91	192.168.0.106	192.168.0.234	OpenFlow	140	Type: OFPT_FLOW_MOD
5132	4.91	192.168.0.106	192.168.0.234	OpenFlow	128	Type: OFPT_FLOW_MOD

Class: OFPMODT_FLOW_MOD (0x0000)
0000 101 = Field: OFPMODT_FLOW_MOD (5)
.....0 = Has mask: False
Length: 2
Value: IPv4 (0x0000)
= OXM field
Class: OFPMODT_FLOW_MOD_BASIC (0x0000)
0001 011 = Field: OFPMODT_FLOW_MOD_BASIC (11)
.....0 = Has mask: False
Length: 4
Value: 192.168.0.114
= OXM field
Class: OFPMODT_FLOW_MOD_BASIC (0x0000)
0001 100 = Field: OFPMODT_FLOW_MOD_BASIC (12)
.....0 = Has mask: False
Length: 4
Value: 192.168.0.170
= Instruction
Type: OFPT_APPLY_ACTIONS (4)
Length: 24
Pad: 00000000
= Action
Type: OFPAT_OUTPUT (0)
Length: 16
Port: 2
Max length: 65535
Pad: 000000000000

Figure 2. Unencrypted OpenFlow messages



No.	Time	Source	Destination	Protocol	Length	Info
5127	4.91	192.168.0.106	192.168.0.234	OpenFlow	510	Type: OFPT_FLOW_MOD
5128	4.91	192.168.0.106	192.168.0.234	OpenFlow	270	Type: OFPT_FLOW_MOD
5129	4.91	192.168.0.106	192.168.0.234	OpenFlow	170	Type: OFPT_FLOW_MOD
5131	4.91	192.168.0.106	192.168.0.234	OpenFlow	140	Type: OFPT_FLOW_MOD
5132	4.91	192.168.0.106	192.168.0.234	OpenFlow	128	Type: OFPT_FLOW_MOD

Class: OFPMODT_FLOW_MOD (0x0000)
0000 101 = Field: OFPMODT_FLOW_MOD (5)
.....0 = Has mask: False
Length: 2
Value: IPv4 (0x0000)
= OXM field
Class: OFPMODT_FLOW_MOD_BASIC (0x0000)
0001 011 = Field: OFPMODT_FLOW_MOD_BASIC (11)
.....0 = Has mask: False
Length: 4
Value: 192.168.0.114
= OXM field
Class: OFPMODT_FLOW_MOD_BASIC (0x0000)
0001 100 = Field: OFPMODT_FLOW_MOD_BASIC (12)
.....0 = Has mask: False
Length: 4
Value: 192.168.0.170
= Instruction
Type: OFPT_APPLY_ACTIONS (4)
Length: 24
Pad: 00000000
= Action
Type: OFPAT_OUTPUT (0)
Length: 16
Port: 2
Max length: 65535
Pad: 000000000000

Figure 3. OpenFlow messages encrypted with TLS1.2

Figures 2 and 3 show OpenFlow messages captured by Wireshark running on a Kali Linux device connected to the same subnet as the Ryu controller. If bad actors manage to sniff OpenFlow packets for a sufficient time period, they could obtain a very detailed topology of the SDN network. Figure 2 reveals a single flow instruction with the source IP address of the

⁸ <https://github.com/mininet/mininet/blob/master/examples/natnet.py>

⁹ <https://techandtrains.com/2014/04/27/open-vswitch-with-ssl-and-mininet/>

BC node, the destination IP of the end-user device and the switch interface this flow should be forwarded on. In contrast, Figure 3 does not reveal any detailed information of the OpenFlow message because the flow is encrypted with TLS version 1.2.

Every such flow update, when sent to OvS switches, could also be forwarded to the BC for registration with the SC. Subsequently, the data relating to these flows stored in the BC could be queried when required. Because the information stored in the BC is immutable, i.e. it cannot be altered or deleted, the BC offers a reliable source of historical data stored in distributed peer nodes. To ensure that encryption of OpenFlow messages is enforced, it is required to generate an SSL certificate for each server or OvS switch and pre-load the certificate to the device before connecting it to the SDN network.

There are two different certificate types involved in this proposed solution. The first pre-loaded one is required for TLS encryption and the second self-signed certificate is used for second-factor authentication. The SSL certificate needs to be signed by the relevant Certificate Authority (CA) and uploaded to the device prior to connecting the device to the SDN network. This ensures that the OpenFlow messages are encrypted from the outset by TLS version 1.2. Without these three files uploaded to the device, i.e. device certificate, private key and the CA certificate, the device would be unable to connect and communicate with the controller

However, OvS switches could be spoofed even when TLS is enforced. In the event that a bad actor manages to obtain the SSL certificate of the victim device, such a spoofed device appears to the controller as a genuine device, provided that the other details, such as the IP address, MAC address and switch ID are also spoofed. For the purposes of the project, a Kali Linux virtual device was configured with identical switch ID, interfaces and MAC addresses and after the spoofed switch acquired identical IP address as the victim device from the DHCP server, it was accepted by the controller. In a lab environment the same certificates were uploaded to the spoofed device as the victim device had.

Flow Manager									
Switch ID(s)					Switch Desc				
#2485379006465					Mfr Desc : Nicira, Inc. Hw Desc : Open vSwitch Sw Desc : 2.17.1 Serial Num : None Dp Desc : None				
Port Desc									
STATE	PORT NO	PEER	NAME	MAX SPEED	HW ADDR	CURR SPEED	CURR CO	TX PACKETS	TX ERRORS
1	0		eth0	1000000	e4:5f:01:57:05:2f	1000000	10248	0	1762
LOCAL	0		br0	0	02:42:ac:22:00:01	0	0	0	609
2	0		eth2	1000000	20:7b:d5:1a:06:14	10000	8193	0	0

Figure 4: Genuine OvS details as seen by the controller

Flow Manager									
Switch ID(s)					Switch Desc				
#2485379006465					Mfr Desc : Nicira, Inc. Hw Desc : Open vSwitch Sw Desc : 3.3.1 Serial Num : None Dp Desc : None				
Port Desc									
STATE	PORT NO	PEER	NAME	MAX SPEED	HW ADDR	CURR SPEED	CURR CO	TX PACKETS	TX ERRORS
4	1	0	eth0	1000000	e4:5f:01:57:05:2f	1000000	102	344	0
LOCAL	0		br0	0	02:42:ac:22:00:01	0	0	184	0
1	2	0	eth2	1000000	20:7b:d5:1a:06:14	10000	8193	0	0

Figure 5: Spoofed device details as seen by the controller

The only difference in the above Figures 4 and 5 is the software version of the OvS switches. With such a spoofed device joining the SDN network, bad actors could manipulate or delete existing flows, which could have serious operational consequences to the network. For these

reason, the following two use-cases were also investigated to attempt to mitigate spoofed switches joining the SDN network.

6.2 Case-study 2: Device registration and validation in the BC

When the device successfully connects to the controller over TLS, the *SDNEther* app verifies if this device is already registered with the SC. If not, the app automatically generates a new self-signed certificate, distributes it to the device and subsequently forwards the copy of this certificate to the SC. This is done via the Secure File Transfer Protocol (SFTP) and the *SDNEther* app uses the imported *Paramiko* library for this purpose. This self-signed certificate is used as a second-factor authentication for the network infrastructure devices such as OvS switches or servers.

The `generate_certificate` method uses `pyOpenSSL` in Python to generate a self-signed certificate.¹⁰ The certificate generated by this method sets the expiration date to two years from its creation. Each certificate generated in this way is unique based on its Common Name (CN) being the IP address of the device, and based on its randomly generated serial number. The certificate is signed with its relevant private key using the SHA-256 hashing algorithm. Because each OvS switch or a server will be subsequently accessed by the app for verification purposes using the IP address rather than the fully qualified domain name (FQDN), specifying the device's IP address in the CN attribute is justified.

The *SDNEther* app then sends such generated certificates to OvS switches or host servers. It uses the `paramiko.SSHClient()` client instance for this purpose. The app uploads the self-signed certificates to remote devices via SFTP, and saves these certificates in the Ubuntu VM local files. Subsequently, the app registers this device's name, its IP address and the certificate in the SC. The fourth data type kept for each device in the SC is the failed login log register outlined in the subsequent use-case. Whenever a server device or an OvS switch re-connects to the SDN network, the app fetches the certificate from the device and compares it with the expected identical certificate stored in the BC. If the certificates are not identical, the network access for the device is blocked by the app that pushes the empty `action[]` instruction (i.e. block all) to the device.

6.3 Case-study 3: Failed login events saved in the BC

SSH servers are enabled on each network infrastructure device like a server or OvS switch connecting to the SDN network. To keep track of failed consecutive login attempts on such devices, a mechanism is in place where after three consecutive failed login attempts such a device sends a curl POST request to the controller over HTTPS. This request message includes the device name, IP address and the message `'three failed login attempts made from {device IP} on {timestamp}'`. Upon receipt of these HTTPS POST requests, the *SDNEther* app uses the ABI and the Web3 connection to append

¹⁰ <https://stackoverflow.com/questions/45873832/how-do-i-create-and-sign-certificates-with-pythons-pyopenssl>

such log messages to the relevant device field in the SC. The administrator can query the historical data of all failed multiple login attempts relating to a given device.

6.4 Case-study 4: End-user device authentication via Metamask

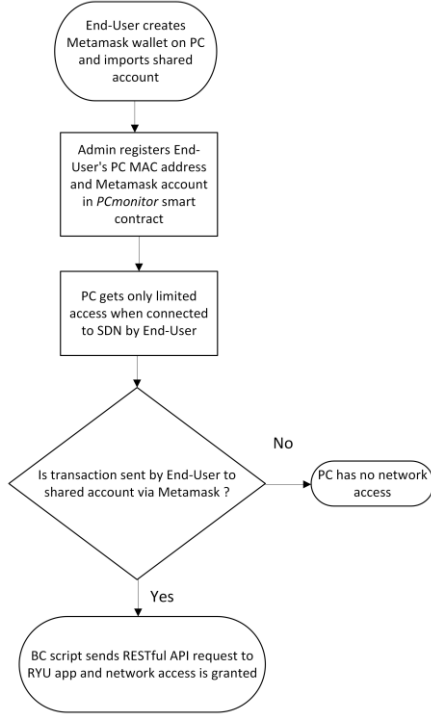


Figure 6. End-user device authentication with BC and Metamask.

End-User devices are initially granted limited access to reach only the IP address of the BC designated node. End-Users need to create a Metamask wallet on the PC requiring SDN network access. Once the Metamask wallet and end-user account have been created, the administrator registers the PC's MAC address and the newly created end-user's Metamask wallet account in the *PCmonitor* smart contract. Subsequently, the end-users connect their PCs to the SDN switch port and send a transaction from their Metamask wallet account to the designated shared account of the BC miner node. A Python script, running alongside the BC, monitors this designated shared account for any transactions. When a transaction is received, the script queries the *PCmonitor* SC for the MAC address of the sender's account and initiates an HTTPS POST request, sending a JSON with transaction details to the app.

Upon receipt of the request, the app pushes new flows to the OvS switch, and as a result, network access for the relevant device is granted. Figure 6 outlines the process for obtaining network access with the help of Metamask, BC monitoring script and the *PCmonitor* SC. Because each account created by BC wallets is unique and created on the basis of the PKI infrastructure, it is virtually impossible to impersonate such accounts without obtaining the private key to such an account. Therefore, this solution, although strong in terms of transaction non-repudiation, might be potentially vulnerable to a phishing attack when a bad actor obtains the Metamask secret login pass phrase, the Metamask password, or the private key of the account created by the end-user in Metamask.

Table 2: Summary statistics for all stages of the authentication process (*in seconds*)

From Start of Metamask transaction till it is detected by the BC monitoring script				
mean	median	standard deviation	minimum duration	maximum duration
0.4444880799243325	0.4015249013900757	0.2576397895844407	0.2443537712097168	1.4556472301483154
From transaction detection by BC monitoring script till HTTPS POST request is detected by the App				
mean	median	standard deviation	minimum duration	maximum duration
0.05731898232510215	0.052414774894714355	0.019496967922950192	0.026781558990478516	0.12189531326293945
From the APP receiving the HTTPS POST request till the flows are pushed to OVS switch				
mean	median	standard deviation	minimum duration	maximum duration
0.004556731173866673	0.0030477046966552734	0.003900135973887843	0.003900135973887843	0.014222145080566406

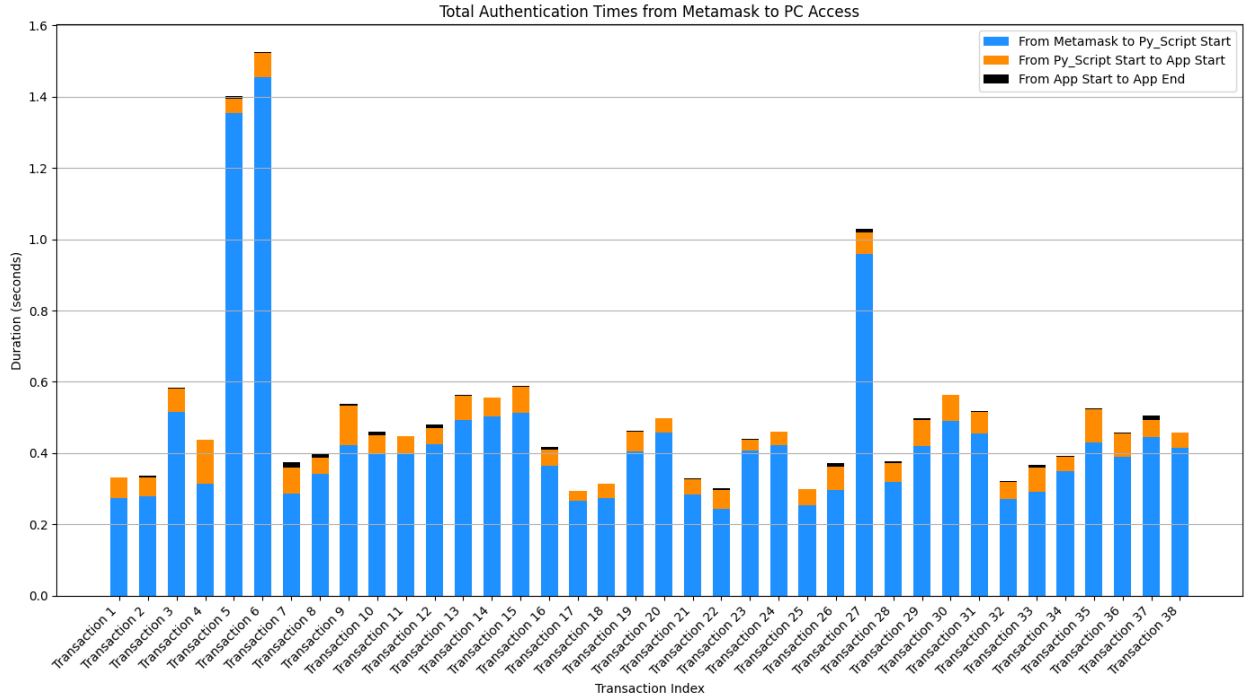


Figure 7: Stacked bar chart showing the total authentication times for the end-user

Table 2 and Figure 7 above present the summary statistics of the end-user authentication times. It takes significantly more time to complete the first stage from Metamask to transaction detection by the BC monitoring script. Compared to the first stage, the second stage (from the detection of transactions by the BC script till HTTPS POST requests are received by the App) is much shorter, and the final stage, resulting in granting network access, is rapid.

6.5 Discussion

Case-study 1 lays the foundations for securing the OpenFlow communications between the controller and the data-plane devices. Without strong TLS encryption in place, the SDN network would be exposed to network sniffing, device spoofing and MITM attacks. It was demonstrated that even with TLS encryption implemented, spoofing attacks could be possible if an attacker obtains the legitimate SSL certificate from the victim device. In such a case a spoofed device is accepted by the controller even with TLS in place. Thus, for this reason further investigation was performed with Case-studies 2 and 3. In Case-study 2 a second-factor authentication was implemented in the form of a unique self-signed certificate generated for each infrastructure device by the app and distributed automatically to the device and to the BC. Subsequently, whenever a device is re-connected to the network, the app triggers the check in which the certificate is fetched from the connecting device and compared with the one stored for it in the BC. The SC performs the comparison and based on its feedback the device is granted network access or blocked completely. A successful test was carried out to simulate tampering with the device SFA certificate, in that making a small change in the certificate resulted in the device failing the BC check and being blocked by the *SDNEther* app.

To reduce the risk of bad actors obtaining management access via SSH to the infrastructure device, a log of failed consecutive logins stored in the BC was proposed in Case-study 3. It was found to be working as expected. However, the design in this case study should go further and implement a notification mechanism informing the administrator of failed login attempts in real-time. An email notification should be sent to the administrator each time three consecutive failed login attempts occur, and then such an event should be also logged in the BC.

Finally, Case-study 4 proved effective and efficient in utilising Metamask combined with the BC and the Python monitoring script to authenticate the end-user devices. Compared to the experiments performed by Petcu et al. [10] the average authentication times recorded in Case-study 4 were **0.506** second compared to **2.698** seconds in the experiment performed by Petcu et al. Although the conclusions in both experiments had some common elements, in that the most time-consuming stage in the authentication process was from the end-user transaction initiation until the handover to the back-end for authentication handling, such a direct comparison needs to be viewed with significant reservation because it is unclear how the end-user interactions with Metamask to initiate authentication were measured by Petcu et al. In Case-study 4 of this project it was attempted to reduce the human factor variability to the minimum by capturing the first Unix-format timestamp at the very end of the transaction when the end-user clicks the final ‘*Confirm*’ button to initiate the transaction. To ensure reliable timestamp synchronisation, the recorded timestamps are all from the same Ubuntu VM, i.e. although the end-user Metamask transaction is performed from a separate PC to where the Ryu app and the BC are running, the JavaScript snippet injected into the Firefox Web Console fetches the timestamp from the Ubuntu VM instead of capturing the timestamp from the local PC. This ensures that at all the stages of the authentication process the timestamps are captured from a single source. Furthermore, the summary statistics indicate that the BC is not causing bottlenecks and the transactions are received and processed efficiently by the BC. The difference in average authentication times between the two experiments might also be attributed to the use of the more efficient PoA consensus mechanism [19] in this project.

Finally, a multi-controller architecture, an important aspect of mitigating scalability issues and the SDN controller being a single point of failure, were out of scope for this project. It is acknowledged that single-controller architectures would widen the surface of potential attacks, particularly DoS and DDoS attacks. It is possible to stage different types of DoS/DDoS attacks targeted primarily at the controller and mitigation against such attacks must also be in place when securing SDN networks. However, due to the fact that a variety DoS/DDoS attack types and mitigation solutions have already been explored [20][21], a separate project dedicated to DoS/DDoS in the context of BC-SDN is suggested.

7 Conclusion and Future Work

This research project attempted to highlight the benefits of integrating the BC and SDN technologies to enhance the access control of data-plane and end-user devices. Four case-studies were implemented and evaluated. The findings of the first case-study indicated that securing the data plane communications with encryption is of paramount importance in mitigating risks, such as data interception, network sniffing, devices spoofing and MITM attacks. It was demonstrated with a Kali Linux VM that a spoofed OvS switch was accepted by the controller, even when TLS encryption was in place, which may lead to devastating results for the entire SDN network. Therefore, enforcing TLS encryption is not a sufficient mitigation measure and second-factor authentication would reduce the risk of potential attacks. The Ethereum SCs and BCs are able to enhance such authentication. Case-study 2 demonstrated that the SCs can act not only as data ledgers but they are capable of comparing and validating SFA certificates and sending their feedback to the app, off-loading the app in workload processing in this way. It was also demonstrated in Case-study 3 that the BC is an excellent ledger for storing logs that cannot be tampered with or deleted due to the immutable characteristic of BCs and their decentralised design. Finally, Case-study 4 demonstrated that end-user device authentication could be efficiently enhanced by the BC, SCs and digital wallets for which the PKI forms the foundation. The BC combined with the digital wallets ensures data integrity and non-repudiation of authentication transactions. Therefore, it would be extremely difficult, if not impossible, to impersonate an end-user without access to Metamask secret pass phrase, account private key, or the password. However, a sophisticated spear phishing attack staged on an individual could potentially obtain such secrets. To conclude, it has been confirmed in numerous studies, and in this project's findings that BCs can enhance the SDN networks in all three aspects of the CIA triad, through private/permissioned BCs architectures, built-in cryptography and transaction anonymity (Confidentiality), through data immutability and tamper-resistance (Integrity), and through distributed/decentralised architectures of BCs (Availability). The future work in respect of BC-SDN integration might explore new aspects of Artificial Intelligence (Machine Learning and its subset - Deep Learning), Moving Target Defence solutions and designing next generation protection mechanisms against controller hijacking attacks.

References

- [1] B. B. Gupta, G. M. Perez, D. P. Agrawal, and D. Gupta, Eds., *Handbook of Computer Networks and Cyber Security*. Cham: Springer International Publishing, pp. 341-387, 2020
- [2] A. Rahman, M. J. Islam, S. S. Band, G. Muhammad, K. Hasan, and P. Tiwari, "Towards a blockchain-SDN-based secure architecture for cloud computing in smart industrial IoT," *Digital Communications and Networks*, Nov. 2022
- [3] A. Rahman, A., Montieri, A., Kundu, D., Karim, Md.R., Islam, Md.J., Umme, S., Nascita, A. and Pescapé, A., "On the Integration of Blockchain and SDN: Overview, Applications, and Future Perspectives," *Journal of Network and Systems Management*, vol. 30, no. 4, Sep. 2022
- [4] K. S. Goud and S. R. Gidituri, "Security Challenges and Related Solutions in Software Defined Networks: A Survey," *International Journal of Computer Networks and Applications*, vol. 9, no. 1, p. 22, Feb. 2022
- [5] B. Agborubere and E. Sanchez-Velazquez, "OpenFlow Communications and TLS Security in Software-Defined Networks," *IEEE Xplore*, Jun. 01, 2017
- [6] P. Ohri and S. G. Neogi, "Software-Defined Networking Security Challenges and Solutions: A Comprehensive Survey," *International Journal of Computing and Digital Systems*, vol. 12, no. 1, pp. 383-400, Jul. 2022
- [7] A. H. Abdi, L. Audah, M. A. Alhartomi, S. Ahmed, and A. Tahir, "Security Control and Data Planes of SDN: A Comprehensive Review of Traditional, AI and MTD Approaches to Security Solutions," *IEEE Access*, pp. 1-1, Jan. 2024.
- [8] T. Alharbi, "Deployment of Blockchain Technology in Software Defined Networks: A Survey," *IEEE Access*, vol. 8, pp. 9146-9156, 2020
- [9] W. Meng, W. Li, and J. Zhou, "Enhancing the security of blockchain-based software defined networking through trust-based traffic fusion and filtration," *Information Fusion*, vol. 70, pp. 60-71, Jun. 2021
- [10] A. Petcu, B. Pahontu, M. Frunzete, and D. A. Stoichescu, "A Secure and Decentralized Authentication Mechanism Based on Web 3.0 and Ethereum Blockchain Technology," *Applied Sciences*, vol. 13, no. 4, p. 2231, Feb. 2023
- [11] M. Latah and K. Kalkan, "DPSec: A blockchain-based data plane authentication protocol for SDNs," *Second International Conference on Blockchain Computing and Applications (BCCA)*, 2020, pp. 22-29.
- [12] A. Derhab, M. Guerroumi, M. Belaoued, and O. Cheikhrouhou, "BMC-SDN: Blockchain-Based Multicontroller Architecture for Secure Software-Defined Networks," *Wireless Communications and Mobile Computing*, vol. 2021, pp. 1-12, Apr. 2021
- [13] M. Steichen, S. Hommes, and R. State, "ChainGuard - A firewall for blockchain applications using SDN with OpenFlow," 2017 Principles, *Systems and Applications of IP Telecommunications (IPTComm)*, 2017
- [14] Z. A. El Houda, L. Khoukhi, and A. Hafid, "ChainSecure - A Scalable and Proactive Solution for Protecting Blockchain Applications Using SDN," *2018 IEEE Global Communications Conference (GLOBECOM)*, Dec. 2018
- [15] P. K. Sharma, S. Singh, Y. S. Jeong, and J. H. Park, "DistBlockNet: A Distributed Blockchains-Based Secure SDN Architecture for IoT Networks," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 78-85, 2017
- [16] J. Hu, M. Reed, N. Thomos, M. F. Al-Naday, and K. Yang, "Securing SDN controlled IoT Networks Through Edge-Blockchain," *IEEE Internet of Things Journal*, pp. 1-1, 2020

- [17] Safi Faizullah, M. Asad. Khan, A. Alzahrani, and I. Khan, "Permissioned Blockchain-Based Security for SDN in IoT Cloud Networks," *International Conference on Advances in the Emerging Computing Technologies (AECT)*, Feb. 2020
- [18] M. B. Jimenez, D. Fernandez, J. E. Rivadeneira, L. Bellido, and A. Cardenas, "A Survey of the Main Security Issues and Solutions for the SDN Architecture," *IEEE Access*, vol. 9, pp. 122016–122038, 2021
- [19] S. R. Fahim, S. Rahman, and S. Mahmood, "Blockchain: A Comparative Study of Consensus Algorithms PoW, PoS, PoA, PoV," *International Journal of Mathematical Sciences and Computing*, vol. 9, no. 3, pp. 46–57, Aug. 2023
- [20] R. Jmal, W. Ghabri, R. Guesmi, B. M. Alshammari, A. S. Alshammari, and H. Alsaif, "Distributed Blockchain-SDN Secure IoT System Based on ANN to Mitigate DDoS Attacks," *Applied Sciences*, vol. 13, no. 8, p. 4953, Jan. 2023
- [21] R. F. Ibrahim, Q. Abu Al-Haija, and A. Ahmad, "DDoS Attack Prevention for Internet of Thing Devices Using Ethereum Blockchain Technology," *Sensors*, vol. 22, no. 18, p. 6806, Sep. 2022