

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL  
INSTITUTO DE INFORMÁTICA  
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANDREI CORDOVA AZEVEDO

**Extracting and Identifying Blockchain  
Transactions in Software-Defined Networks**

Work presented in partial fulfillment  
of the requirements for the degree of  
Bachelor in Computer Science

Advisor: Prof. Dr. Lisandro Zambenedetti  
Granville

Coadvisor: Dr. Eder John Scheid

Porto Alegre  
February 2024

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos André Bulhões

Vice-Reitora: Prof<sup>ª</sup>. Patricia Pranke

Pró-Reitora de Graduação: Prof<sup>ª</sup>. Cíntia Inês Boll

Diretora do Instituto de Informática: Prof<sup>ª</sup>. Carla Maria Dal Sasso Freitas

Coordenador do Curso de Ciência de Computação: Prof. Marcelo Walter

Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

## **ACKNOWLEDGMENTS**

Only in Brazilian Portuguese. See page 4.

## **AGRADECIMENTOS**

Agradeço à Universidade Federal do Rio Grande do Sul, pela formação de excelência proporcionada, e aos professores e servidores que contribuíram com a mesma. Ao meu orientador, Professor Doutor Lisandro Zambenedetti, pela oportunidade que me foi dada, e ao meu coorientador, Doutor Eder John Scheid, por todo o empenho no auxílio e nas orientações ao longo deste trabalho.

Gostaria de agradecer a minha família, pelo suporte durante toda a minha formação. Em especial, ao meu pai, Paulo, pelo apoio incondicional durante minha jornada, como aluno e como pessoa, e minha avó, Valda, que sempre me apoiou, e é lembrada com muito carinho. Aos meus mais queridos amigos, que me ajudaram a passar por este caminho (e vários outros) de forma leve e feliz.

## ABSTRACT

The adoption of Blockchain (BC) technology has rapidly grown, expanding its applications to different fields such as finance, cybersecurity, and the Internet of Things. Thus, identifying BC traffic in a network could enable the implementation of specific performance and Quality-of-Service (QoS) requirements derived from the wide range of BC applications. This work aims to perform an in-depth analysis of Ethereum packets in an emulated private BC using a Lightweight SDN Testbed (LST), a testbed tool for network research that relies on Docker containers and Software-defined Networking (SDN). It also explores the possibilities of expanding the network topology in the present work for future research on BCs. The results obtained from the analysis performed in the proposed scenario in this work demonstrate that identifying Ethereum transactions is a challenging task, given its encryption. However, the novel dataset generated in this work, containing both Ethereum and regular HTTP traffic, can be explored to perform automated traffic classification in real time and provide a base for future research on Ethereum packet identification using Machine Learning (ML) algorithms. Further, it is shown that, if given enough resources, the use of LST can be easily extended to different network topologies in BC emulation.

**Keywords:** Blockchain. Ethereum. SDN. Docker containers.

## **Extraindo e Identificando Transações Blockchain em Redes Definidas por Software**

### **RESUMO**

A adoção da tecnologia Blockchain (BC) tem crescido rapidamente, expandindo as suas aplicações a diferentes domínios, como as finanças, a cibersegurança e a Internet das Coisas. Assim, a identificação do tráfego de BC numa rede poderia permitir a implementação de requisitos específicos de desempenho e Qualidade de Serviço (QoS) derivados da vasta gama de aplicações de BC. Este trabalho tem como objetivo realizar uma análise aprofundada dos pacotes Ethereum em um BC privado emulado usando um Lightweight SDN Testbed (LST), uma ferramenta de testbed para pesquisa de rede que se baseia em contêineres Docker e Redes Definidas por Software (SDN). Também explora as possibilidades de expandir a topologia de rede no presente trabalho para futuras pesquisas sobre BCs. Os resultados obtidos a partir da análise realizada no cenário proposto neste trabalho demonstram que a identificação de transações Ethereum é uma tarefa desafiante, dada a sua encriptação. No entanto, o novo conjunto de dados gerado neste trabalho, que contém tráfego HTTP regular e Ethereum, pode ser explorado para efetuar a classificação automática do tráfego em tempo real e fornecer uma base para investigação futura sobre a identificação de pacotes Ethereum utilizando algoritmos de Aprendizagem de Máquina (ML). Além disso, é demonstrado que, se forem dados recursos suficientes, a utilização do LST pode ser facilmente alargada a diferentes topologias de rede na emulação BC.

**Palavras-chave:** Blockchain, Ethereum, Redes Definidas por Software, Containers Docker.

## LIST OF FIGURES

Figure 2.1 Node discovery process in Ethereum.....	14
Figure 2.2 DEVP2P protocol traffic flow in Ethereum .....	15
Figure 2.3 Communication between Execution (eth1 client) and Consensus (eth2 client) layers.....	16
Figure 2.4 Separation of Data plane and Control plane in SDN.....	19
Figure 4.1 Emulated Scenario with LST.....	27
Figure 4.2 Dataset Row Example.....	36
Figure 5.1 Sending a transaction using Geth through LST.....	38
Figure 5.2 Retrieving details from a transaction using Geth through LST.....	39
Figure 5.3 Sending several transactions from any of the four participating nodes with different ETH values.....	40
Figure 5.4 Ethereum Wireshark Dissector used in node discovery messages .....	40
Figure 5.5 Ethereum traffic analysis .....	41
Figure 5.6 Structure of a transaction message in Ethereum.....	42
Figure 5.7 Variation of packet size in regular HTTP traffic.....	43
Figure 5.8 Optimal number of cluster analysis using Elbow Method.....	44
Figure 5.9 KMeans Clusterization of Ethereum TCP Traffic .....	45
Figure 5.10 KMeans Clusterization of Ethereum UDP Traffic.....	46

## LIST OF TABLES

Table 3.1 Related Work Comparison .....	24
---	----



## LIST OF ABBREVIATIONS AND ACRONYMS

BC	Blockchain
CNN	Convolutional Neural Network
DApps	Distributed Applications
DPI	Deep Packet Inspection
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
EIP	Ethereum Improvement Proposal
EOA	Externally Owned Account
ETH	Ether
IoT	Internet of Things
LST	Lightweight SDN Testbed
ML	Machine Learning
EVM	Ethereum Virtual Machine
P2P	Peer-to-Peer
PCAP	Packet Capture
PoA	Proof-of-Authority
PoS	Proof-of-Stake
QoS	Quality-Of-Service
RLP	Recursive Length prefix
SDN	Software Defined Networking
SVM	Support Vector Machines
UTXO	Unspent Transaction Output
Wasm	WebAssembly
WCSS	Within-Cluster Sum of Squares

## CONTENTS

<b>1 INTRODUCTION</b> .....	<b>11</b>
<b>2 BACKGROUND</b> .....	<b>13</b>
<b>2.1 Ethereum</b> .....	<b>13</b>
2.1.1 Execution Layer .....	13
2.1.2 Consensus Layer .....	16
2.1.3 Blocks .....	17
2.1.4 Transactions .....	17
<b>2.2 Software-Defined Networking</b> .....	<b>18</b>
<b>2.3 LST</b> .....	<b>20</b>
<b>3 RELATED WORK</b> .....	<b>22</b>
<b>3.1 State-of-the-Art Approaches</b> .....	<b>22</b>
<b>3.2 Comparison and Discussion</b> .....	<b>24</b>
<b>4 ANALYZING AND IDENTIFYING ETHEREUM TRANSACTIONS</b> .....	<b>26</b>
<b>4.1 Methodology</b> .....	<b>26</b>
<b>4.2 Scenario</b> .....	<b>27</b>
<b>4.3 Implementing the Scenario</b> .....	<b>28</b>
4.3.1 Deploying a Private Ethereum Blockchain .....	28
4.3.2 Creating Ethereum Nodes .....	29
4.3.3 SDN Controller .....	31
4.3.4 Sending Transactions .....	33
4.3.5 Generating HTTP Traffic .....	33
4.3.6 Capturing Packets and Dataset Creation.....	35
<b>5 EVALUATION AND DISCUSSION</b> .....	<b>37</b>
<b>5.1 Emulating an Ethereum Network</b> .....	<b>37</b>
<b>5.2 Traffic Analysis</b> .....	<b>39</b>
5.2.1 Separation and categorization of Ethereum traffic.....	39
5.2.2 Categorization and Evaluation of Transaction Packets .....	41
<b>5.3 Challenges and Remarks</b> .....	<b>45</b>
<b>6 SUMMARY, CONCLUSION AND FUTURE WORK</b> .....	<b>47</b>
<b>REFERENCES</b> .....	<b>49</b>
<b>APPENDIX A — PUBLISHED PAPER – ERRC 2023</b> .....	<b>52</b>

## 1 INTRODUCTION

Blockchain (BC) is a technology based on the concept of distributed ledgers that stores information in a decentralized and distributed network (SCHEID et al., 2021). The information is persisted as a transaction, that is validated and stored in a chain of blocks on the distributed network. Its structure is based on Peer-to-Peer (P2P) communication, which evicts the necessity of a central authority and thus enables the existence of a decentralized network with shared resources (BELOTTI et al., 2019). It was first proposed in 2008, as the database and means to solve the double-spending problem for the Bitcoin cryptocurrency (NAKAMOTO, 2009), and since then, its adoption has expanded to several other areas such as finances, cybersecurity and Internet of Things (IoT) (MONRAT; SCHELÉN; ANDERSSON, 2019), as well as to other emerging cryptocurrencies, such as Ethereum (BUTERIN, 2014).

In this sense, from the wide range of possible applications of BC technology and different BCs, derives different Quality-Of-Service (QoS) and performance requirements (MONRAT; SCHELÉN; ANDERSSON, 2019) Thus, being able to identify BC-related packets in a network and differ them from regular traffic would be beneficial as this could allow performing several different actions such as *(i)* identifying and prioritizing BC applications traffic, *(ii)* detecting devices that are possibly infected with cryptojackers and *(iii)* performing load-balancing for different BC nodes.

The employment of Machine Learning (ML) techniques in the field of BC traffic analysis has been a common practice in recent scientific literature. Algorithms such as KMeans allows a deeper analysis of possible patterns and relationships existent in BC traffic, providing the possibility of visualizing such attributes. Thus, clusterizing BC traffic data with the use of KMeans can be helpful in the visual identification of patterns related to BC packets and their behavior.

Since the architecture of BCs is complex, comprising several different layers (*e.g.*, network layer, data model layer, execution layer and application layer) (BELOTTI et al., 2019), it is difficult to create and maintain an actual BC environment. Further, the size of BCs, such as Bitcoin and Ethereum, is over 400 GB, requiring dedicated hardware (SANKA; CHEUNG, 2021). This characteristic hinders the use of real BC environments for research purposes, specially since it makes replicating test scenarios even more unfeasible. Fortunately, it is possible to resort to BC emulation and simulators to avoid such difficulties. Simulation aims to reproduce a BC system model, enabling its evalua-

tion in a parameterized way, without the need to implement the entire system (PAULAVIČIUS; GRIGAITIS; FILATOVAS, 2021) whereas emulation aims to replicate the behavior of a system as closely as possible (GILL; LEE; QIAO, 2021).

Thus, in this work, the Lightweight SDN Testbed (LST) (KAIHARA et al., 2022) tool is used to emulate an Ethereum-based BC network containing different nodes. Several Ethereum transactions and normal traffic requests are sent and captured by the Software-Defined Networking (SDN) controller inside LST. The network traffic is then inspected, and different analysis using ML regarding Ethereum transactions characteristics and its differences from regular traffic are brought upon. The contributions of this work are summarized as follows:

- A novel dataset containing both Ethereum and normal HTTP traffic, properly labeled, that can be used for further research on Ethereum traffic analysis;
- A real-world scenario using the LST tool that emulates an Ethereum BC network in a feasible and highly configurable manner; and

The remainder of this work is organized as follows: Chapter 2 describes the background on BC, Ethereum, and its implementation, as well as the SDN concept. Chapter 3 discusses related work on Ethereum BC traffic analysis and the use of SDN in such a context. Chapter 4 describes the methodology and the implementation of the emulation scenario. Followed by Chapter 5, which details the experiments, presents the results and their evaluations, and discusses challenges and future research on the subject. Finally, Chapter 6 concludes this work presenting conclusions and future work.

## 2 BACKGROUND

This chapter details the concepts, paradigms, and technologies involved in this work. Hence, the main components of the Ethereum BC are described, followed by SDN and the LST tool.

### 2.1 Ethereum

Ethereum was proposed in 2015 as a BC able to support the creation of versatile applications (BUTERIN, 2014). It offers, within its BC ecosystem, the support for developers to create Decentralized Applications (DApps) using a Turing-complete programming language. With such a language, developers define immutable BC-based smart contracts that contain specific rules for the enforcement of transactions used by DApps. Thus, by providing this language, Ethereum became the *De Facto* standard for DApps.

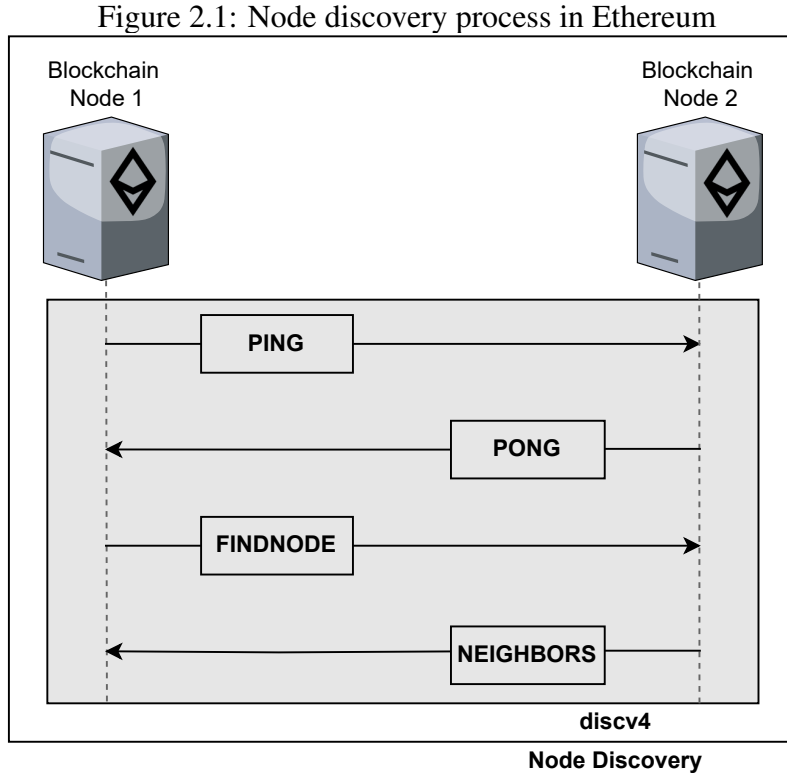
Unlike Bitcoin, where the BC state is defined by Unspent Transaction Output (UTXO) transactions, Ethereum uses account-based states, where each account is defined by a set of fields (*e.g.*, nonce, balance, contract code and storage) (BUTERIN, 2014), and state transitions are transactions of information or value between different accounts.

The Ethereum networking layer is a stack of protocols that define rules for communication between participating nodes, allowing them to discover each other and exchange information. It can be subdivided into two different layers: (i) the Execution layer and (ii) the Consensus layer (ethereum.org, 2023). While execution clients send transactions to other nodes in the execution layer, consensus layer clients propagates proposed chain blocks across the network. Thus, it is necessary to have two different P2P networks: one for connecting execution clients and other for consensus clients.

#### 2.1.1 Execution Layer

Execution clients must be able to discover other peers in the network to connect and later exchange information between them. For such, the execution layer provides the discovery stack `discv4`, based on UDP protocol. Figure 2.1 depicts message flow in the node discovery process. New nodes connect to boot nodes, which are initial nodes whose addresses are hardcoded in the node client, through PING-PONG messages. A PING

message informs the bootnode of the existence of a new node, who replies with a PONG message. The new node can then request a list of peers to perform connection through a FINDNODE request to the bootnode.



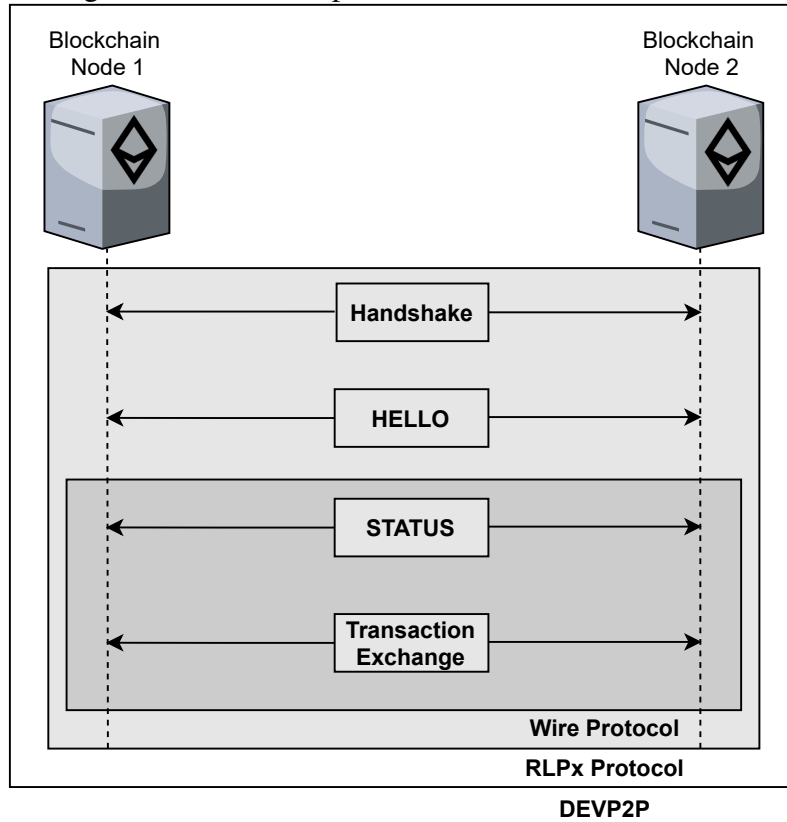
Source: (The Author, 2024)

After the discovery process, communication between the new node and existing nodes are governed by *DEVP2P*, a stack of network protocols based on TCP that defines how peers participating in the Ethereum network should structure their communication. Figure 2.2 demonstrates the communication process under *DEVP2P* protocol stack. *RLPx* is the protocol inside the *DEVP2P* stack that defines how peers must establish and maintain a connection. Messages in the *RLPx* protocol are encoded using Recursive Length Prefix (RLP) and encrypted through the *secp256k1* Elliptic Curve Integrated Encryption Scheme (ECIES), an asymmetric encryption method. Thus, initializing a connection between two Ethereum nodes consists on performing a cryptographic handshake through the *RLPx* protocol, allowing participating nodes to agree on the ECIES ephemeral key used for further encrypted communication. A successful handshake triggers both nodes to exchange HELLO messages, confirming that the connection was established and allowing subsequent transaction exchange communication through the Wire subprotocol (ethereum.org, 2023).

The Wire subprotocol defines the structure of messages used to synchronize trans-

action pools between connected peers. Nodes perform this exchange of pending transactions in order to enable miners to pick new transactions and insert them into the BC.

Figure 2.2: DEVP2P protocol traffic flow in Ethereum



Source: (The Author, 2024)

## 2.1.2 Consensus Layer

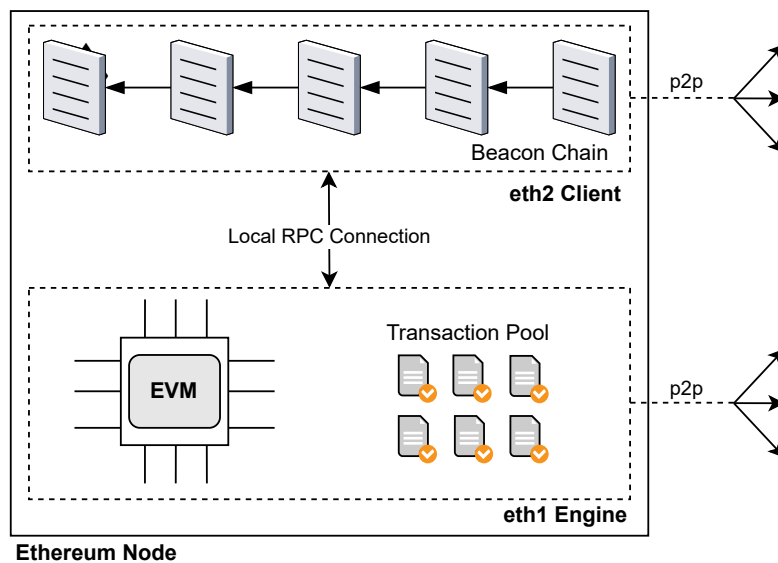
Similarly to the Execution layer, the Consensus layer also provides a discovery stack so that consensus clients can connect amongst each other and perform block propagation. The main difference from the discovery stack present in the Execution layer, is that the one present here uses discovery stack `discv5`. While it also relies on UDP, this protocol takes advantage of `LIBP2P` network stack instead of `DEV2P` used in the Execution layer. Thus, `RLPx` sessions are not present in the Consensus layer, as `LIBP2P` uses noise secure channel handshakes instead.

Ethereum relies on the Proof-of-Stake (PoS) consensus method, where a node must stake capital (*i.e.*, Ethereum coins known as Ether - ETH), into a smart contract. Only nodes with stake in the smart contract can propose and validate blocks; if the node does not follow a set of rules or propose invalid blocks, its stake is reduced, decreasing its chance of proposing blocks.

Block propagation in the Consensus layer is performed through the use of `LIBP2P`'s gossip protocol `gossipsubv1` (DEV2P Community, 2023). Consensus clients must ensure that received blocks are valid by checking its metadata and sender identification.

Figure 2.3 describes how Execution and Consensus clients, which run in parallel, are connected. Both clients exchange information between each other using a local RPC connection, and they both maintain a separate P2P network to connect and communicate with other consensus and execution clients from different peers.

Figure 2.3: Communication between Execution (eth1 client) and Consensus (eth2 client) layers



Source: (The Author, 2024)



### 2.1.3 Blocks

In Ethereum, a block is a structure that contains batches of transaction information. Each block is linked to the previous one by pointing to a hash that is cryptographically derived from the block data, preventing any possible attempts of fraud by changing blocks content. This structure is used to create a history of transactions in Ethereum, where participating peers agree and synchronize on the committed transactions.

To create and preserve this history, all blocks and the transactions inside of each of them are ordered. A validator node is randomly selected to be a block proposer on the network, and must bundle transactions together in a block structure to propagate it to other peers and propose a new global state. The receiving peers must then check for the correctness of the proposed block and decide if they agree with the new state. As Ethereum uses PoS protocol, this means that validators must also stake ETH into contracts when performing these operations in order to prevent dishonest or illicit behavior in the BC (ethereum.org, 2023).

### 2.1.4 Transactions

A transaction in Ethereum represents a cryptographically signed instruction that updates the global state of the BC. It can be initiated by an Externally Owned Account (EOA), and can also interact with smart contracts, for executing an already deployed contract or deploying a new contract code. Each transaction must be broadcasted in the network in order to validator nodes execute and insert them into a new block, changing the global state (ethereum.org, 2023).

A transaction object exchanged by peers in Ethereum consists of the following attributes (BUTERIN, 2014):

- `nonce`: a sequence number that is used to avoid message replay. This number is increment at every new transaction sent by the address;
- `gas-price`: In order to complete a transaction, the sender must provide a gas value to cover its computational cost. This amount is used to pay miners per computational step, and is represented in WEI, which is the smallest denomination of ETH. One ETH is equivalent to  $1e^{18}$  WEI.
- `gas-limit`: the maximum number of gas units that a transaction can consume.

Each computational operation in the Ethereum Virtual Machine (EVM) has a gas cost associated to them (BUTERIN, 2014).

- `recipient`: a 20-byte Ethereum address, which can be an EOA or a contract;
- `value`: the amount of Ether to be sent to the recipient. This value is also denoted in WEI.
- `data`: binary data payload. This field is used when interacting with smart contracts, either to deploy a function or to send it as an input to an existing contract.
- `V`, `R` and `S`: components of the Elliptic Curve Digital Signature Algorithm (ECDSA). `V` represents the recovery ID of the ECDSA, while `R` and `S` are outputs of the ECDSA.

The state transition function in Ethereum checks if the transaction is well-formed (*i.e.*, verifies the correctness of the signature and if the nonce matches the one in the sender's account) (VUJIČIĆ; JAGODIĆ; RANĐIĆ, 2018). Upon validating the transaction, the sender's nonce is incremented and the gas allocated for the transaction is removed from their balance. If the transaction is executed completed, any unused gas amount is refunded to the sender, and a new global state of the Ethereum network is created.

## 2.2 Software-Defined Networking

Traditional IP networks are usually configured by network operators using complex low-level commands directly to the proprietary control software that runs inside routers and switches to enable the application of different network policies to traffic. Besides that, the coupling of control and data planes inside network traffic devices increases the difficulty of implementing highly customizable network policies (KREUTZ et al., 2014). Thus, it requires high effort and operation cost to manage traditional networks in a flexible manner.

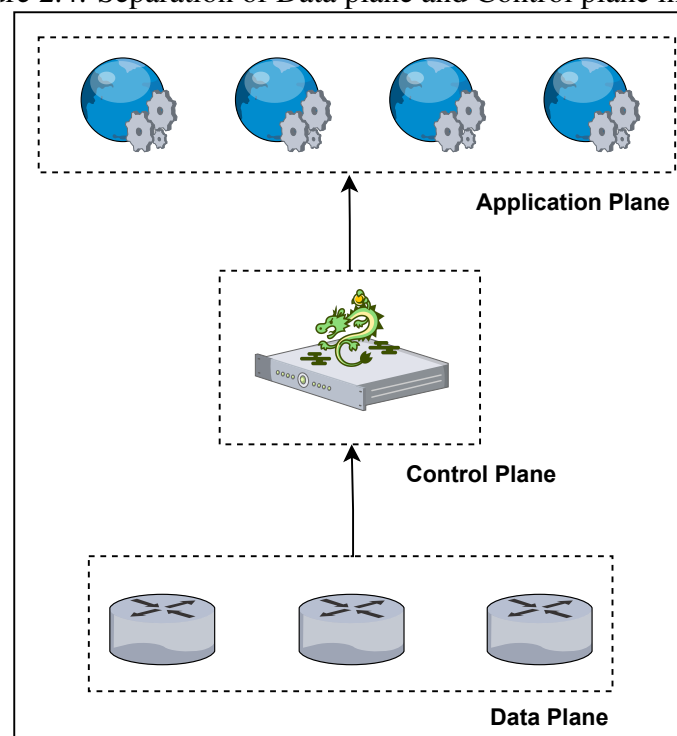
Such difficulties drove the interest of making computer networks more flexible in configuration. Over the time, several contributions have been made in order to achieve a high level of flexibility in computer networks configurations, such as *(i)* introducing programmable functions in networks, *(ii)* decoupling the control and data planes and *(iii)* the proposal of OpenFlow Application Programming Interface (API) (FEAMSTER; REX-FORD; ZEGURA, 2014).

OpenFlow offers an interface to program flow-tables in different switches and

routers, in a vendor-independent approach. It puts together several different capabilities that are common to most Ethernet routers and switches and exposes them through an API to enable users to control flows in a scalable and feasible manner (MCKEOWN et al., 2008). Flows are packet-handling rules to match traffic and perform different possible actions, such as forwarding, dropping and modifying packets. Routers and switches have their own flow-tables, which is a collection of these rules. Thus, OpenFlow enables, by exposing an open protocol through an API, that users program these flow-tables in different routers and switches, without having to look up to vendor-specific implementations.

All these previous contributions have fostered the development of SDN, which is a programmable network that uses software-based controllers to communicate with hardware and network traffic. Its main characteristics that differs it from a traditional network are that *(i)* it separates the control plane from the data plane, *(ii)* it consolidates the control plane, thus making a single controller responsible for the data plane elements through an OpenFlow API, transforming network switches into simple forwarding devices and *(iii)* forwarding decisions are based on flows instead of destination (FEAMSTER; REXFORD; ZEGURA, 2014). This allows to perform real-time traffic management based on user defined policies in an extensible manner, as it is easier to manage network configurations through SDN than of regular routers.

Figure 2.4: Separation of Data plane and Control plane in SDN



Source: (The Author, 2024)

As SDN offers a centralized control while exchanging information between different network layers, it arises the possibility of tackling several common network performance-related issues by implementing proper algorithms on the software controller, such as QoS support, congestion control and data traffic scheduling (XIA et al., 2014).

### 2.3 LST

LST (KAIHARA et al., 2022) is a lightweight and easy-to-use tool for SDN and security studies. It allows reproducing scenarios in the context of SDN through the virtualization of physical infrastructures, by taking advantage of Docker containers. Through the virtualization offered by Docker containers, it is possible to achieve a highly configurable and isolated environment.

The tool relies on the Ryu Controller (Ryu SDN Framework Community, 2017), a SDN framework for network management and control applications that supports several protocols, such as OpenFlow. Thus, it allows to perform different actions on the virtual switches *e.g.*, network analysis, changing network route policies and identifying malicious attacks. Further, as it relies on Docker containers, such a controller can be replaced and modified in a flexible manner.

Listing 2.3 presents the LST code used to emulate a simple topology composed of two hosts (one Webserver and one client), one switch, and one SDN controller). The configuration of the hosts containers is performed in Lines 8 and 9, where they are instantiated in LST with a given Docker image. The same occurs for the SDN controller on Line 11, and for the switch, which is instantiated in Line 10 but using the default Docker image supplied by LST. Hosts are connected to the switch on Lines 13 to 15, and the switch is then connected to the SDN controller on Line 16. IP addresses for each container are set on Lines 18 to 22. The switch points to the controller on Line 25, which is initialized on Line 24. All of the containers on the topology point to the switch on Lines 26 to 31, and the later is connected to the internet with a given IP address on Line 27.

Listing 2.1 – Initializing nodes using LST

```

1 signer = Host('signer')
2 h1 = Host('node1')
3 h2 = Host('node2')
4 s1 = Switch('s1')
5 c1 = Controller('c1')
6
7 signer.instantiate(dockerImage='eth-node')
```

```
8 h1.instantiate(dockerImage="eth-node")
9 h2.instantiate(dockerImage="eth-node")
10 s1.instantiate()
11 c1.instantiate(dockerImage="eth-controller")
12
13 signer.connect(s1)
14 h1.connect(s1)
15 h2.connect(s1)
16 s1.connect(c1)
17
18 signer.setIp('10.1.1.1', 24, s1)
19 h1.setIp('10.1.1.2', 24, s1)
20 h2.setIp('10.1.1.3', 24, s1)
21 s1.setIp('10.1.1.4', 24)
22 c1.setIp('10.1.1.5', 24, s1)
23
24 c1.initController('10.1.1.5', 9001)
25 s1.setController('10.1.1.5', 9001)
26
27 s1.connectToInternet('10.1.1.6', 24)
28 signer.setDefaultGateway('10.1.1.6', s1)
29 h1.setDefaultGateway('10.1.1.6', s1)
30 h2.setDefaultGateway('10.1.1.6', s1)
31 c1.setDefaultGateway('10.1.1.6', s1)
```

### 3 RELATED WORK

There are efforts on capturing and identifying BC transactions in SDN available in the literature. This chapter details such approaches and compares them with the approach proposed in this work.

#### 3.1 State-of-the-Art Approaches

(TEKINER; ACAR; ULUAGAC, 2022) offers an cryptojacking detecting mechanism for IoT devices based on ML algorithms that are trained using network features such as packets per second and average packet size. The proposed methodology is applied to in-browser and host-based cryptojacking malware. Although it does differentiate regular traffic from cryptojacking-related, it does not apply to regular BC traffic. Therefore it is not possible to use the proposed tool to capture and identify BC transactions that are not related to malware.

Similarly, (CAPROLU et al., 2021) proposes a ML-based framework for detection of cryptominers on Bitcoin, Bytecoin and Monero BCs. It relies on metrics such as interarrival time and packet size to train the model and subsequently perform traffic classification, though it does analyzes Ethereum BC traffic.

(RODRIGUEZ; POSSEGA, 2018) monitors resource-related API calls and CPU consumption by browsers to identify malicious cryptojackers in a host. The work uses Support Vector Machines (SVM) to train a model on a dataset of malicious and benign websites and classify them. Its application is specific to the context of browser-based cryptojackers and thus does not tackle on identifying specific BCs traffic on a network.

(MUÑOZ; SUÁREZ-VARELA; BARLET-ROS, 2021) uses NetFlow protocol, which performs flow-level network measurements, to feed data to different ML models to identify cryptominers on Bitcoin, BitcoinCash, DogeCoin, Litecoin and Monero BCs.

(NING et al., 2019) detects in-browser malicious cryptojackers of Monero BC using a Convolutional Neural Network (CNN) approach, monitoring features such as CPU, memory and disk usage. The proposed tool is constrained to detection of Monero-related malicious malware in browsers, thus not being possible to identify regular Ethereum packets with it.

On a similar approach, (KELTON et al., 2020) proposes a browser-based tool for detecting cryptomining activities in web pages. It analyzes the behavior of CPU, memory

and disk usage, representing it as a timeseries, and correlates it to patterns that indicates cryptojacking activity using a CNN algorithm. The work extends the detection to Monero, Webchain and uPlexa BCs, but still lacks an identification of possible Ethereum transaction activities.

(KIM et al., 2021) uses a semi-supervised learning approach with AutoEncoder to detect anomalies in real-world Bitcoin traffic data. Bitcoin BC network traffic in this work is classified through a set of eleven features, which includes average byte size of messages, clock cycles cost per second for a node and the number of received messages. Given that the work does not make use of SDN, it does not extend to take advantage of its capabilities such as blocking such anomalies in the network nor does it focus on Ethereum traffic, though it could be finetuned to apply to other cryptocurrencies since it relies on standard network features as described by the authors.

(NASEEM et al., 2021) proposes Minos, a lightweight tool for detecting cryptojacking based on WebAssembly (Wasm) modules in web pages. Feeding both malicious and benign Wasm binaries to a CNN-based ML model, the tool classifies cryptomining activity through Wasm modules on the network in real-time. The work is restricted to the context of Wasm and in-browser detection, thus not being possible to extend it to other cryptocurrencies such as Ethereum.

(RUSSO; SRNDIĆ; LASKOV, 2021) reconstructs Stratum protocol from raw Net-Flow records and use its data on a One-class classifier ML algorithm to detect cryptomining of Monero BC. It detects encrypted mining traffic since it relies on network metadata instead of payload data and on features that describe the behavior of the Stratum protocol.

(NETO et al., 2020) proposes an incremental learning model that detects and blocks cryptocurrency mining flows on top of SDN controllers. Similarly to our approach, it leverages SDN with the use of Ryu Controller, although its implementation is not specific to Ethereum, and it uses Mininet as an emulation environment, thus lacking the advantages of using LST as a tool for such context.

(CABAJ; GREGORCZYK; MAZURCZYK, 2018) uses HTTP traffic characteristics such as message sequence and content size to classify packets of crypto ransomware and proposes a SDN-based detection system. While it does also leverage SDN, the authors propose a method based on HTTP communication characteristics of two common ransomware families, thus not being applicable to the Ethereum BC.

(GABA et al., 2022) offers a ML model prototype for detection of security attacks on Ethereum BC with the use of SDN. It relies on a dataset of historical Ethereum Classic

transactions, contracts, blocks and logs to extract relevant features to the model. The proposed method is specific to the context of security attacks, *i.e.*, it does not tackle on specific Ethereum BC traffic characteristics and differentiating it from regular network traffic.

Other works such as (NING et al., 2019) and (KELTON et al., 2020) focus on metrics such as CPU, memory and disk usage to evaluate and classify whether a packet is related to BC traffic or not in their respective ML models. This is not applicable in a SDN controller as it does not have access to such metrics from the nodes connecting to it, thus being not feasible to identify Ethereum packets in such manner.

### 3.2 Comparison and Discussion

Table 3.1 summarizes and compares the related work on the field in different dimensions, such as which BC is analyzed, the identification method employed, which metrics are used, and if the approach relies on SDN or not.

Table 3.1: Related Work Comparison

Work	Blockchain	Identification Method	Metrics	SDN	Environment
(TEKINER; ACAR; ULUAGAC, 2022)	Not Specified	KNN SVM RF	Timestamps Packet size	✗	Datasets
(NETO et al., 2020)	Not Specified	RF GBT	Network traffic	✓	Emulated
(CAPROLU et al., 2021)	Bitcoin Bytecoin Monero	RF	Inter-arrival time Packet size	✗	Emulated
(RODRIGUEZ; POSSEGA, 2018)	Not Specified	SVM	CPU usage	✗	Datasets
(MUÑOZ; SUÁREZ-VARELA; BARLET-ROS, 2021)	Bitcoin BitcoinCash DogeCoin LiteCoin Monero	SVM CART Naive Bayes	NetFlow traffic data	✗	Datasets
(NING et al., 2019)	Monero	CapsNet	CPU Memory Disk usage	✗	Datasets
(KELTON et al., 2020)	Monero Webchain uPlexa	CNN	CPU Memory Network usage	✗	Datasets
(KIM et al., 2021)	Bitcoin	AutoEncoder	Packet size Number of packets	✗	Datasets
(CABAJ; GREGORCZYK; MAZURCZYK, 2018)	Not Specified	HTTP Traffic	POST messages size	✓	Emulated
(GABA et al., 2022)	Ethereum Classic	Not Specified	Not Specified	✓	Datasets
(NASEEM et al., 2021)	Not Specified	CNN	WASM binaries	✗	Datasets
(RUSSO; SRNDIĆ; LASKOV, 2021)	Monero	OCC	NetFlow traffic data	✗	Datasets
(MUÑOZ, 2019)	Bitcoin	SVM CART Naive Bayes	NetFlow traffic data	✗	Datasets

Based on the related work research, most of the available work tries to capture and identify BC-related traffic in traditional networks. (NETO et al., 2020), (CABAJ; GREGORCZYK; MAZURCZYK, 2018) and (GABA et al., 2022) are the only ones who



takes advantage of SDN when trying to identify BC traffic. Besides, it is possible to notice that the most common approach is using Machine Learning algorithms to analyze traffic and classify it. As data is typically encrypted in BCs traffic to ensure security and privacy, Deep Packet Inspection (DPI) approaches tend to not be very feasible when trying to detect and classify BC-related packets in real time.

It is also possible to notice that there are several works on identifying illicit browser-based cryptomining activity, specially on Monero BC, as it is one of the most mined digital coins (RUSSO; SRNDIĆ; LASKOV, 2021). Therefore, there are few works that proposes characterizing and identifying other BCs traffic data like Ethereum.

Thus, this work focuses on trying to identify packets belonging to Ethereum traffic and differentiate them from regular network traffic, by leveraging LST tool with an underlying SDN and a Ryu Controller, as this has not been addressed yet in the current available scientific literature.

## 4 ANALYZING AND IDENTIFYING ETHEREUM TRANSACTIONS

This chapter details the methodology applied in the present work for the identification and analysis of Ethereum-related packets using LST, the scripts used to send Ethereum transactions amongst nodes in SDN, and the generated dataset of Ethereum-related traffic.

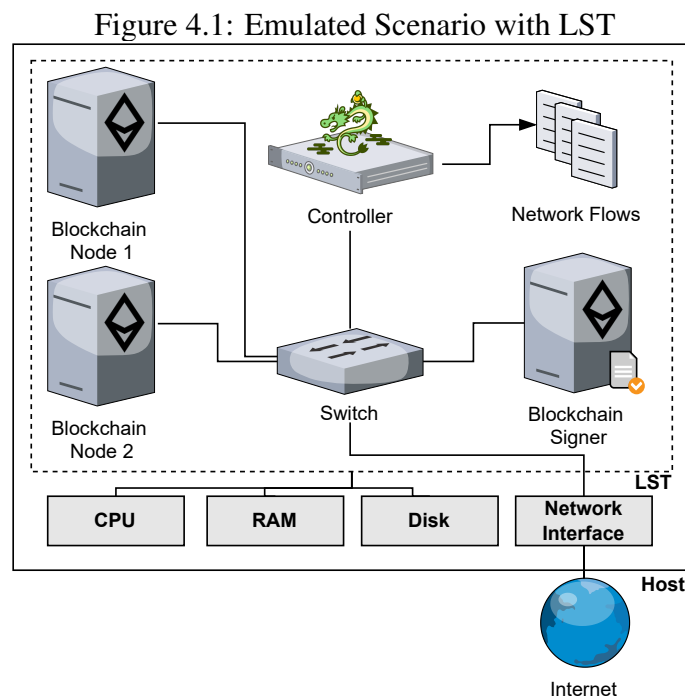
### 4.1 Methodology

In this work, we defined a methodology to investigate the network behavior of Ethereum BC transactions in comparison with regular HTTP traffic. The methodology included several steps, each contributing to the overall aim of the study. The methodology was structured to not only set up a controlled emulated BC environment but also to capture and analyze network traffic data. This approach enabled us to create a unique dataset and conduct the analysis of the traffic patterns. The steps of such a methodology included:

1. **Deploying a Private Ethereum BC:** A private Ethereum network was deployed using a custom genesis file, defining the initial state of the BC and the balance of participating accounts;
2. **Creating Ethereum Nodes:** A set of nodes was created and deployed in the private BC to conduct analysis of the communication between them;
3. **Using the SDN Ryu Controller to Capture Packets:** Packet Capture (PCAP) files were saved directly from the controller for posterior processing, generating a novel dataset with both regular HTTP and Ethereum traffic;
4. **Sending Ethereum transactions:** Participating nodes in the private BC performed several transactions with different values in order to enable posterior analysis of its packets;
5. **Generating random HTTP traffic:** To compare the differences between Ethereum and regular HTTP traffic, a random traffic generator was used to include its packets into the final dataset; and
6. **Analysis of patterns and characteristics:** Ethereum-related packets were dissected to understand their characteristics and differences towards regular HTTP traffic.

## 4.2 Scenario

Figure 4.1 illustrates the proposed scenario. In the scenario, a private Ethereum BC was created, relying on the Proof-of-Authority (PoA) consensus protocol, where one or more nodes are assigned as validators for the entire network. One of the participating nodes is defined as the Ethereum Block Signer. This node contains the genesis file for the private Ethereum BC system, which defines the initial data on the first block in the chain as well as other configurations (*e.g.*, difficulty, gas limit, initial account balances and minimum block time in seconds). Then, two other generic BC nodes, containing the same genesis file, are initialized and later connected to the virtual switch.



Source: (The Author, 2024)

The virtual switch container is instantiated with a Ryu Controller that saves incoming packets from the network into a PCAP file. This enables later inspection of the emulated BC system network traffic. It is important to mention that the controller can be customized and exchanged if required; thus, showing that LST can serve as a testbed for experiments regarding the analysis of traffic in different scenarios.

Once all of the participating nodes are connected to the virtual switch, they are instantiated using Geth ([geth.ethereum.org](https://geth.ethereum.org/), 2024), the official Ethereum client built in Go. After the node discovery process ends and the nodes are synchronizing with BC network, it is possible to start sending transactions to the network, check account balances, and inspect network flows that are recorded through the Ryu Controller.

### 4.3 Implementing the Scenario

This section details the implementation of the scenario, including related BC files, Docker files, and LST configuration.

#### 4.3.1 Deploying a Private Ethereum Blockchain

The private Ethereum BC network in the present work is deployed using Geth. A customized genesis file is used to define the initial state of the BC, as well as other core settings (*i.e.*, current chain ID, gas limit in blocks and the starting balances of listed accounts). Listing 4.1 illustrates the genesis file for the initial state of the BC.

Listing 4.1 – Configuration file for the private Ethereum BC

```

1 { "config": {
2   "chainId": 3107,
3   "homesteadBlock": 0,
4   "byzantiumBlock": 0,
5   "constantinopleBlock": 0,
6   "petersburgBlock": 0,
7   "istanbulBlock": 0,
8   "berlinBlock": 0,
9   "clique": {
10    "period": 5,
11    "epoch": 30000
12  }
13 },
14 "difficulty": "1",
15 "gasLimit": "8000000",
16 "alloc": {
17   "0x6b4F3286fe87612e7Deb71A2FBedA0e948Ad4980": {
18     "balance": "1000000000000000000"
19   },
20   "0x0b913e0F6093819aff423254AaA8cAd82FDa9b02": {
21     "balance": "5000000000000000000"
22   },
23   "0x84564ba949a198f5e0f09bfe7233760F29d3a1d0": {
24     "balance": "5000000000000000000"
25   }
26 } }

```

The *chainId* (Line 3) is used to identify the current chain. It is a unique value, preventing replay in the BC; *gasLimit* (Line 19) is the block gas limit and is usually defined to minimize transaction and propagation time by limiting the amount of gas that the set of transactions inside a block can consume; *clique* (Line 13) is the PoA consensus

protocol in the private BC. The minimum difference between timestamps of two consecutive blocks is defined in seconds as *period*, and *epoch* refers to the number of blocks after which there will be a reset on the pending votes for the proposed blocks (Lines 14 and 15, respectively). The three participating nodes in the proposed scenario are assigned with a starting amount of ETH under the attribute array *alloc* (Line 20) so that they can perform transactions, allowing posterior capture of Ethereum transaction packets in LST with the help of Ryu SDN Controller for analysis.

### 4.3.2 Creating Ethereum Nodes

Setting up a new node on the private BC through Geth consists of using the `geth init` command, passing the genesis file as an input parameter. Additionally, a command with different parameters is executed, mainly to (i) unlock the account in order to be able to send transactions without the need of manually approving them, (ii) provide the account password and (iii) provide the bootnode list so that the new node can execute the node discovery process. Listing 4.2 demonstrates the shell script containing the commands used with Geth to create a new Ethereum node on LST.

Listing 4.2 – Initialization commands executed on Geth to create a new regular Ethereum Node

```

1 geth init --datadir /home/.ethereum/data /home/.ethereum/genesis.json
2 geth --nat=extip:$1 --datadir=/home/.ethereum/data --networkid=3107 \
3 --keystore=/home/.ethereum/keystore --unlock "0x0b913e0F6093819aff423254AaA8cAd82FDa9b0
  2" \
4 --allow-insecure-unlock --password "/home/.ethereum/account-password" \
5 --http --http.port=8545 --http.corsdomain="*" --http.api=net,admin,eth \
6 --bootnodes enode://2adeac6710220735cf6c4737e752644b93a4102ea388e77c3196666326cebc68bfd0
  2472630e300018a22c3e9952d09d915e29c693ca4dcd9231e3407d86b9c4@10.1.1.1:30303 \
7 --verbosity=5 > /home/geth.log 2>&1

```

The `unlock` parameter receives an account address (Line 3) to allow it to perform transactions through Geth. `bootnode` list parameter (Line 6) points to the signer account address, as it will be the initial node in this proposed scenario, allowing future participating nodes to connect and inquire for other nodes that are connected on the private BC.

Creating the signer node is done through the same steps. The difference is that this node receives the `mine` and `miner.etherbase` parameters that refers to, respectively, enable the mining process and thus being able to propose new blocks, and the public address for the block mining rewards. Listing 4.3 illustrates the shell script used to set up

a new signer node on the private BC.

#### Listing 4.3 – Initialization commands executed on Geth to create a new signer Ethereum Node

```

1 geth init --datadir /home/.ethereum/data /home/.ethereum/genesis.json
2 geth --nat=extip:$1 --datadir=/home/.ethereum/data --networkid=3107 \
3 --mine --miner.etherbase=0x6b4F3286fe87612e7Deb71A2FBedA0e948Ad4980 \
4 --keystore=/home/.ethereum/keystore --unlock "0x6b4F3286fe87612e7Deb71A2FBedA0e948Ad498
  0" \
5 --allow-insecure-unlock --password "/home/.ethereum/account-password" \
6 --http --http.port=8545 --http.corsdomain="*" --http.api=miner,net,admin,eth \
7 --verbosity=5 > /home/geth.log 2>&1

```

The `miner.etherbase` parameter receives the signer account address (Line 3). As this node will be used as the initial node on the private BC, it is not necessary to provide it with the `bootnode` parameter as shown in Listing 4.2.

Both signer and regular nodes are deployed as Docker containers in LST. Therefore, it is possible to achieve an isolated environment for each node, that is also independent from the operating system that is running LST, contributing to a replicable test scenario. Listing 4.4 presents the Dockerfile used to initialize a container with all the necessary files (*e.g.*, the genesis files presented in Listing 4.1 is copied to the node in Line 18) for the deployment of an Ethereum node in the scenario's private BC.

#### Listing 4.4 – Dockerfile used for boot of a new Ethereum node on the private BC in LST

```

1 FROM ubuntu:20.04
2
3 ENV DEBIAN_FRONTEND noninteractive
4
5 RUN apt-get update \
6 && RUNLEVEL=1 apt-get install -y cron samba openssh-server sudo net-tools iproute2
  iputils-ping iptables nano apt-utils
7
8 RUN apt-get update \
9 && RUNLEVEL=1 apt-get install -y software-properties-common \
10 && add-apt-repository -y ppa:ethereum/ethereum \
11 && apt-get update \
12 && apt-get install -y ethereum
13
14 RUN apt-get install -y tcpdump
15
16 RUN mkdir /home/.ethereum
17 # copy genesis.json
18 COPY genesis.json /home/.ethereum/
19
20 # copy premade keys
21 COPY keystore/ /home/.ethereum/keystore
22
23 # copy password for keys

```

```

24 COPY account-password /home/.ethereum/
25
26 COPY command-signer.sh /home
27 COPY command-node.sh /home
28
29 COPY onboot.sh /home
30 RUN chmod +x /home/onboot.sh
31 CMD ["/home/onboot.sh"]
32
33 # For connecting via Open SSL
34 EXPOSE 22
35
36 # Ethereum ports
37 EXPOSE 30303
38 EXPOSE 8545

```

As regular nodes and the signer node use the same Dockerfile to build its image, both initialization scripts presented in Listings 4.2 and 4.3 are copied to the environment (Lines 26 and 27). Password keys are also copied to the container environment (Line 24) so that Geth can unlock the accounts and allow transactions to be performed without the need of manual approval. Once the container is running, the corresponding shell script with Geth commands is executed and the Ethereum node is initialized and connected to the BC.

### 4.3.3 SDN Controller

A custom implementation of Ryu SDN Controller is used in the present scenario. It allows capturing incoming packets in real time, so that they can be saved in a PCAP file for posterior analysis and inspection of Ethereum packets and its differences of regular HTTP traffic packets. Analogously to the deployment of Ethereum nodes, the SDN Controller is also deployed as a Docker container in LST. Thus, a customized Dockerfile is used to perform the deployment of the container in the network with the necessary packages and scripts (*i.e.*, Python, Ryu and the custom Ryu controller implementation that saves incoming packets in a file). Listing 4.5 demonstrates the Dockerfile used to create the SDN Controller container in LST.

Listing 4.5 – Dockerfile used to setup the custom Ryu SDN Controller using LST

```

1 FROM ubuntu:20.04
2 RUN apt update \
3 && RUNLEVEL=1 apt install -y --no-install-recommends sudo net-tools iproute2 iputils-
  ping python3 python3-pip iptables nano\

```

```

4 && python3 -m pip install --upgrade pip \
5 && pip3 install ryu eventlet==0.30.2 pandas \
6 && apt-get -o Dpkg::Options::="--force-confmiss" install --reinstall netbase
7
8 RUN apt-get install -y tcpdump
9
10 COPY controller.py /home
11
12 COPY onboot.sh /home
13 RUN chmod +x /home/onboot.sh
14 RUN touch /home/file.pcap
15 CMD ["/home/onboot.sh"]

```

After booting, the controller container runs the custom Ryu Controller implementation that was copied (Line 10). A PCAP file is created in the container environment (Line 14) in order to save incoming packets from the network using `pcaplib`, a library available from Ryu's Python package (Nippon Telegraph and Telephone Corporation, 2014). Listings 4.6 and 4.7 demonstrates the initialization of the custom Ryu controller class and the handling of incoming packets, respectively.

Listing 4.6 – Initialization of the custom implementation of Ryu SDN Controller class

```

1 def __init__(self, *args, **kwargs):
2     super(SimpleSwitch, self).__init__(*args, **kwargs)
3     self.datapaths = {}
4     self.mac_to_port = {}
5     self.pcap_writer = pcaplib.Writer(open('/home/file.pcap', mode='wb'))

```

Initializing the custom class object used in the scenario consists of, in addition to initializing Ryu's base class (Line 2), instantiating `pcaplib`'s `Writer` class object (Line 5) so that it can afterwards save incoming packets into a given PCAP file.

Listing 4.7 – Handling of incoming packets performed in the custom Ryu SDN Controller

```

1 @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
2 def _packet_in_handler(self, ev):
3     msg = ev.msg
4     datapath = msg.datapath
5     ofproto = datapath.ofproto
6     parser = datapath.ofproto_parser
7     pkt = packet.Packet(msg.data)
8     self.pcap_writer.write_pkt(ev.msg.data)

```

The handling of incoming packets in Ryu is defined by setting a trigger to a specific OpenFlow event of packet income (Line 1). In the declared function, it is possible to perform several different actions directly to the packet (*e.g.*, checking its content, modifying a specific field or even dropping the packet). For the proposed scenario in this work, the only objective is to record received packets for posterior analysis, which is done using



the `write_pkt` function from `pcaplib` library (Line 8).

#### 4.3.4 Sending Transactions

Once the private BC is initialized and all the participating nodes, as well as the SDN controller, are deployed, we can then start sending transactions from a node to an existing account address. To do so, a customized function that executes a command inside of a docker container is used in the experiment. Listing 4.3.4 demonstrates the function created to send transactions amongst participating nodes in the BC.

Listing 4.8 – Customized function to send transactions in the private BC

```

1 def sendTransaction(_node, _from, _to, _value, _gas):
2     try:
3         sendTransactionCommand = '''geth attach --exec 'eth.sendTransaction({from: ""'+
4             _from+''',to: ""'+_to+''', value: '''+str(_value)+''', gas: '''+str(_gas)+'''})'
5             /home/.ethereum/data/geth.ipc'''
6         print(sendTransactionCommand)
7         subprocess.run('''docker exec '''+ _node + ''' ''' + sendTransactionCommand,
8             shell=True)
9         return True
10    except Exception as e:
11        print(e)
12    return False

```

It is possible to pass parameters of value, gas, sender address and receiver address, as well as from which Docker Ethereum node to execute the transaction (Line 1). The container will then run Geth client's `sendTransaction` function with the respective parameters (Line 5), sending the transaction to the private blockchain, which will undergo the process of transaction validation and later insertion into a block through the PoA protocol.

#### 4.3.5 Generating HTTP Traffic

To generate a dataset containing regular HTTP traffic as well as Ethereum-related packets, this work relied on `noisy` (HURY, 2018), a public library that generates random HTTP and DNS traffic noise. It consists of a simple Python script that makes HTTP requests to websites that are listed in a JSON configuration file. Thus, it is possible to define several different sources, which allows making requests that vary in packet size, so that it does not bias later analysis and comparison of Ethereum and regular HTTP

traffic. An example of noisy's configuration file used in the proposed scenario is shown in Listing 4.9.

Listing 4.9 – Configuration file used in noisy

```
1 {
2   "max_depth": 25,
3   "min_sleep": 3,
4   "max_sleep": 6,
5   "timeout": false,
6   "root_urls": [
7     "http://4chan.org",
8     "https://www.reddit.com",
9     "https://www.yahoo.com",
10    "http://www.cnn.com",
11    "http://www.ebay.com",
12    "https://wikipedia.org",
13    "https://youtube.com",
14    "https://github.com",
15    "https://medium.com"
16  ],
17  "blacklisted_urls": [
18    ".css",
19    ".ico",
20    ".xml",
21    ".png",
22    ".iso"
23  ],
24  "user_agents": [
25    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/601.7.7 (KHTML,
26      like Gecko) Version/9.1.2 Safari/601.7.7",
27    "Mozilla/5.0 (iPad; CPU OS 9_3_2 like Mac OS X) AppleWebKit/601.1.46 (KHTML,
28      like Gecko) Version/9.0 Mobile/13F69 Safari/601.1",
29    "Mozilla/5.0 (iPhone; CPU iPhone OS 9_1 like Mac OS X) AppleWebKit/601.1.46 (
30      KHTML, like Gecko) Version/9.0 Mobile/13B143 Safari/601.1"
31  ]
32 }
```

Websites to which requests will be executed are defined under the `root_urls` attribute (Line 6). The library also allows defining a list of blacklisted websites or extensions under `blacklisted_urls` attribute (Line 17), which in this scenario is used to avoid downloading common frontend files to the container environment when running the script, and a list of user agents under `user_agents` attribute (Line 24), so it can simulate requests being made by different users through different web browsers.

### 4.3.6 Capturing Packets and Dataset Creation

In order to collect Ethereum traffic and regular HTTP traffic, two executions of the proposed scenario were performed: one exclusively with Ethereum traffic, and the other exclusively with regular HTTP traffic using `noisy`. This was done so that the packets in the final dataset could be correctly labeled and classified, ensuring that future analysis using ML algorithms and manual inspection would be fed with proper data. Both executions were performed for one hour, resulting in two different PCAP files with similar sizes. As shown in Listing 4.7, received packets in the SDN controller are written in a PCAP file that is saved in the controller environment. As LST takes advantage of Docker container technology, it is possible to get the content of the SDN controller container and save it into the machine running the experiment. Thus, after each execution of the proposed scenario, the final PCAP file for each one was saved for posterior processing.

Since PCAP files contain numerous information that are not relevant to the context of this work, they were post-processed and converted into CSV files containing only selected attributes that seemed possibly the most influential and relevant for an analysis and comparison of Ethereum and regular HTTP traffic. Listing 4.3.6 demonstrates how the PCAP file containing Ethereum traffic obtained from an execution of the proposed scenario is converted into a CSV file so that its data can later be inserted into the final dataset containing both Ethereum and regular network traffic.

Listing 4.10 – Conversion of PCAP file to CSV selecting only relevant attributes

```

1 def readPCAP(filepath):
2     packets = PcapReader(filepath)
3     i = 0
4     packet_list = []
5     for packet in packets:
6         try:
7             if packet.haslayer(TCP):
8                 if len(packet[TCP].payload) > 0:
9                     packet_list.insert(i, (packet[IP].src, packet[TCP].sport, packet[IP]
10                      .dst, packet[TCP].dport, len(packet), bytes(packet[TCP].payload), PROTOCOL.TCP,
11                      LABEL.ETH_TRAFFIC))
12                 else:
13                     packet_list.insert(i, (packet[IP].src, packet[TCP].sport, packet[IP]
14                      .dst, packet[TCP].dport, len(packet), '', PROTOCOL.TCP, LABEL.ETH_TRAFFIC))
15                 elif packet.haslayer(UDP):
16                     if len(packet[UDP].payload) > 0:
17                         packet_list.insert(i, (packet[IP].src, packet[UDP].sport, packet[IP]
18                          .dst, packet[UDP].dport, len(packet), bytes(packet[UDP].payload), PROTOCOL.UDP,
19                          LABEL.ETH_TRAFFIC))
20                 else:

```

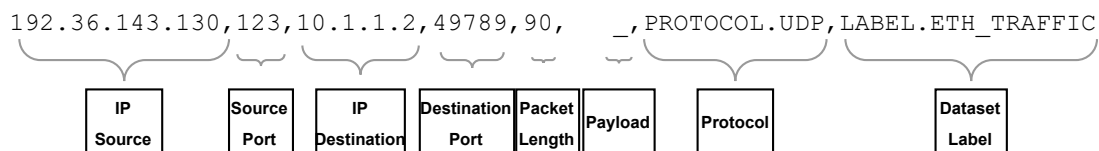
```

16         packet_list.insert(i, (packet[IP].src, packet[UDP].sport, packet[IP
17     ].dst, packet[UDP].dport, len(packet), '', PROTOCOL.UDP, LABEL.ETH_TRAFFIC))
18     except:
19         pass

```

To read and process resulting PCAP files from the scenario, `scapy` (Scapy Community, 2024) library is used in the script. Thus, by declaring an object with a given PCAP file path, it reads its packets into memory (Line 2). TCP and UDP packets are treated separately (Lines 7 and 12) as it is necessary to access corresponding fields for each protocol. The selected attributes were, in order, (i) IP source, (ii) Source port, (iii) IP destination, (iv) Destination port, (v) Packet length, (vi) Packet payload, (vii) Packet protocol and (viii) Dataset label. An example of the resulting dataset can be seen in Figure 4.2.

Figure 4.2: Dataset Row Example



Source: (The Author, 2024)

Thus, after both executions of the proposed scenario with Ethereum traffic and regular HTTP traffic, a novel dataset in CSV format was generated containing rows in the model presented by Figure 4.2, enabling an evaluation and discussion of Ethereum traffic and its differences towards regular HTTP traffic.

## 5 EVALUATION AND DISCUSSION

The experiments were conducted on a host with an Ubuntu 22.04 LTS operating system, with 16 GB RAM and an AMD® Ryzen 5 3600 6-core processor. The source code (*e.g.*, Docker files, BC-related files, and LST scenario definition file) to reproduce the scenario, as well as the final dataset generated in this work, are available at <<https://github.com/andrei-azevedo/extracting-ethereum-traffic-lst>>.

### 5.1 Emulating an Ethereum Network

Deploying a private Ethereum blockchain using LST has been shown to be a very feasible and easy task (AZEVEDO et al., 2023). As LST is as highly configurable and flexible tool that relies on Docker containers technology, it allows to create an isolated environment, which is suitable to replicate test scenarios in the context of network security and SDN. Thus, creating a network topology such as the one defined in the scenario proposed on Figure 4.1 can be done with low effort. Listing 5.1 demonstrates how the network topology on the proposed scenario for this work was deployed using LST. Due to the use of Docker images to deploy each participating node, inserting new nodes into the network topology can be done by simply instantiating a new object with a given Docker image parameter (Lines 2 and 3) and connecting them to the virtual switch (Lines 14, 15 and 16).

Listing 5.1 – Creating the proposed Ethereum blockchain network topology using LST

```

1 signer = Host('signer')
2 h1 = Host('node1')
3 h2 = Host('node2')
4 s1 = Switch('s1')
5 c1 = Controller('c1')
6
7 signer.instantiate(dockerImage='eth-node')
8 h1.instantiate(dockerImage="eth-node")
9 h2.instantiate(dockerImage="eth-node")
10 s1.instantiate()
11 c1.instantiate(dockerImage="eth-controller")
12
13 signer.connect(s1)
14 h1.connect(s1)
15 h2.connect(s1)
16 s1.connect(c1)

```

Figure 5.1 demonstrates an example of 10 transactions (see Line 110) being sent

using the emulated Ethereum BC system using LST code, implementing the proposed network topology depicted in Figure 4.1. In the example, we successfully sent transactions from a node using the function `sendTransaction()` (see line 112) that calls to a given `node1`'s Geth to send the transaction with defined parameters, such as from and to addresses, ETH amount and gas. In the figure, the red dashed box highlights the translated geth command from the `sendTransaction()` function and the transaction hash.

Figure 5.1: Sending a transaction using Geth through LST

```

110 for i in range(10):
111     event.wait(2)
112     sendTransaction('node1', '0x0b913e0F6093819aff423254AaA8cAd82FDa9b02',
113                    '0x84564ba949a198f5e0f09bfe7233760f29d3a1d0',
114                    random.randrange(4000000000000, 5000000000000),
115                    random.randrange(21000, 22000))

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS sudo - blockchain-demo + v [ ] [ ] ... ^ x
INFO [02-01|02:25:20.301] Persisted trie from memory database      nodes=5 size=673.00B t
ime=91.201706ms gcnodes=0 gcsiz=0.00B gctime=0s livenodes=0 livesize=0.00B
INFO [02-01|02:25:21.063] Successfully wrote genesis state      database=lightchaindat
a hash=74e67f..35468f
INFO [02-01|02:25:21.099] Successfully wrote genesis state      database=lightchaindat
a hash=74e67f..35468f
geth attach --exec 'eth.sendTransaction({from: "0x0b913e0F6093819aff423254AaA8cAd82FDa9b0
2",to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 48058332990361, gas: 21967})'
/home/.ethereum/data/geth.ipc
"0xb209065d0a6a9616dc94624577f955a745ead18c807db8d976d273d0ca0001c3"

```

Source: (The Author, 2024)

After sending a transaction using Geth through LST, it is possible to check its details by using the returned transaction hash, as shown in Figure 5.1, demonstrating the correctness of the deployed BC network using LST. Figure 5.2 shows the retrieval of a transaction details using its unique hash value. Lines 118 and 119 demonstrates the execution of Geth's `getTransaction` method in LST passing a transaction hash as parameter.

LST is a very flexible tool that allows us to easily set up new nodes in the blockchain network and submit transactions with different ETH values. Figure 5.3 depicts an example of a different topology containing four common nodes (two more nodes than the original scenario), and randomly sending several transactions with different ETH values among them (see lines 129 to 133).

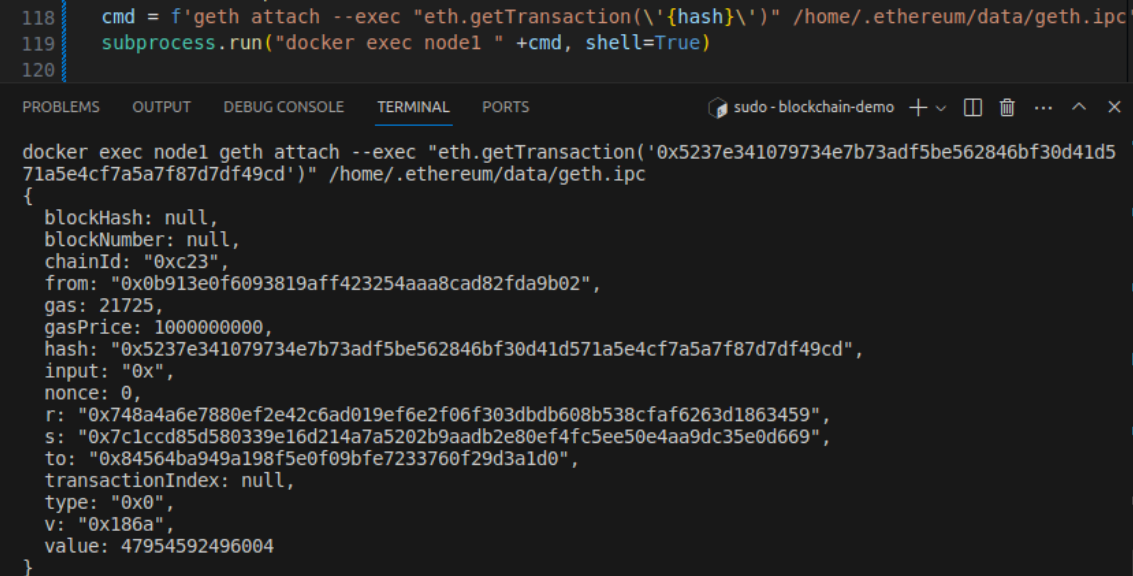
This allows us to verify the communication between several nodes in the network with the signer and verify if all the nodes are synchronizing with the BC. Further, with the use of LST and the experiment defined as a Python script, it can be repeated by other researches to achieve the same result, fostering reproducible research. Moreover, the network topology using LST can be extended up to the available resources on a host's ma-

Figure 5.2: Retrieving details from a transaction using Geth through LST

```

118 cmd = f'geth attach --exec "eth.getTransaction('{hash}')" /home/.ethereum/data/geth.ipc'
119 subprocess.run("docker exec node1 " +cmd, shell=True)
120

```



```

docker exec node1 geth attach --exec "eth.getTransaction('0x5237e341079734e7b73adf5be562846bf30d41d571a5e4cf7a5a7f87d7df49cd')" /home/.ethereum/data/geth.ipc
{
  blockHash: null,
  blockNumber: null,
  chainId: "0xc23",
  from: "0x0b913e0f6093819aff423254aaa8cad82fda9b02",
  gas: 21725,
  gasPrice: 1000000000,
  hash: "0x5237e341079734e7b73adf5be562846bf30d41d571a5e4cf7a5a7f87d7df49cd",
  input: "0x",
  nonce: 0,
  r: "0x748a4a6e7880ef2e42c6ad019ef6e2f06f303dbdb608b538cfaf6263d1863459",
  s: "0x7c1ccd85d580339e16d214a7a5202b9aadb2e80ef4fc5ee50e4aa9dc35e0d669",
  to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0",
  transactionIndex: null,
  type: "0x0",
  v: "0x186a",
  value: 47954592496004
}

```

Source: (The Author, 2024)

chine, thus being possible to emulate complex BCs and other networks in a reproducible manner if given enough resources (*i.e.*, memory to deploy new Docker containers).

## 5.2 Traffic Analysis

The evaluation and analysis of Ethereum traffic and transactions was divided into (i) Separation and categorization of Ethereum traffic and (ii) Evaluating characteristics of transaction packets and its differences towards other Ethereum-related packets and regular HTTP traffic using ML techniques.

### 5.2.1 Separation and categorization of Ethereum traffic

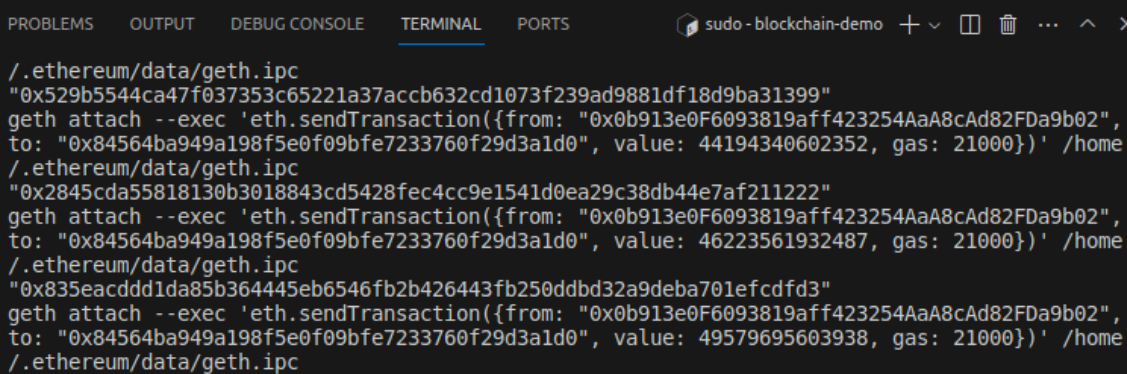
In order to evaluate the characteristics of and to correctly identify transaction traffic in Ethereum, it was necessary to categorize which packets belonged to transactions and which didn't. To do so, all of the private Ethereum BC traffic from the executions of the proposed scenario in this work was extracted and recorded into PCAP files for analysis. It is not feasible to manually identify and categorize specific packets and relate them to transactions exclusively through payload analysis, specially due to the encryption used in Ethereum, which avoids the use of DPI techniques to inspect packet payload and other relevant attributes, as shown in the available literature (GABA et al., 2022). There

Figure 5.3: Sending several transactions from any of the four participating nodes with different ETH values

```

126 nodeList = ['node1', 'node2', 'node3', 'node4']
127 for i in range(10):
128     randomNode = random.choice(nodeList)
129     randomETHValue = random.randint(40000000000000, 50000000000000)
130     sendTransaction('node1', '0x0b913e0F6093819aff423254AaA8cAd82FDa9b02',
131                    '0x84564ba949a198f5e0f09bfe7233760f29d3a1d0',
132                    randomETHValue,
133                    21000)

```



```

/ .ethereum/data/geth.ipc
"0x529b5544ca47f037353c65221a37accb632cd1073f239ad9881df18d9ba31399"
geth attach --exec 'eth.sendTransaction({from: "0x0b913e0F6093819aff423254AaA8cAd82FDa9b02",
to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 44194340602352, gas: 21000})' /home
/ .ethereum/data/geth.ipc
"0x2845cda55818130b3018843cd5428fec4cc9e1541d0ea29c38db44e7af211222"
geth attach --exec 'eth.sendTransaction({from: "0x0b913e0F6093819aff423254AaA8cAd82FDa9b02",
to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 46223561932487, gas: 21000})' /home
/ .ethereum/data/geth.ipc
"0x835eacd1da85b364445eb6546fb2b426443fb250ddb32a9deba701efcdfd3"
geth attach --exec 'eth.sendTransaction({from: "0x0b913e0F6093819aff423254AaA8cAd82FDa9b02",
to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 49579695603938, gas: 21000})' /home
/ .ethereum/data/geth.ipc

```

Source: (The Author, 2024)

are some characteristics in specific protocols used in Ethereum that can facilitate the partial identification of its packets, *i.e.*, the fact that node discovery process uses `discv4` subprotocol, which runs in UDP, as depicted in Figure 2.1. Thus, we have relied on a Wireshark Ethereum Dissector (bcsec.org, 2018) to categorize node discovery packets and separate them from the rest of Ethereum traffic to facilitate posterior categorization of transaction packets. Figure 5.4 shows an example of `discv4` packets identified in Wireshark on the PCAP files obtained from the proposed scenario using the dissector.

Figure 5.4: Ethereum Wireshark Dissector used in node discovery messages

613	13.124840	10.1.1.1	65.21.206.66	DEVP2P	176	30303	→	37523	Len=134	(PING)
614	13.125107	10.1.1.1	188.192.105.49	DEVP2P	213	30303	→	30403	Len=171	(FindNode)
618	13.197818	10.1.1.1	108.165.154.214	DEVP2P	176	30303	→	30311	Len=134	(PING)
625	13.292226	54.159.61.97	10.1.1.1	DEVP2P	199	30305	→	30303	Len=157	(PONG)
626	13.292538	54.159.61.97	10.1.1.1	DEVP2P	176	30305	→	30303	Len=134	(PING)
634	13.372415	188.192.105.49	10.1.1.1	DEVP2P	1099	30403	→	30303	Len=1057	(Neighbors)
635	13.374483	188.192.105.49	10.1.1.1	DEVP2P	467	30403	→	30303	Len=425	(Neighbors)

Source: (The Author, 2024)

It is possible to see that the node on IP address `10.1.1.1` requests for the private BC bootnode for other nodes through the `FindNode` message, as explained on Section 2.1.1 regarding the node discovery process. The bootnode then proceeds to answer with messages of type `Neighbors`. Hence, it is possible to categorize packets belonging to the node discovery process under the `discv4` subprotocol, which facilitates the posterior analysis and categorization of remaining Ethereum-related packets.

As the rest of the Ethereum communication is performed through the `devp2p`

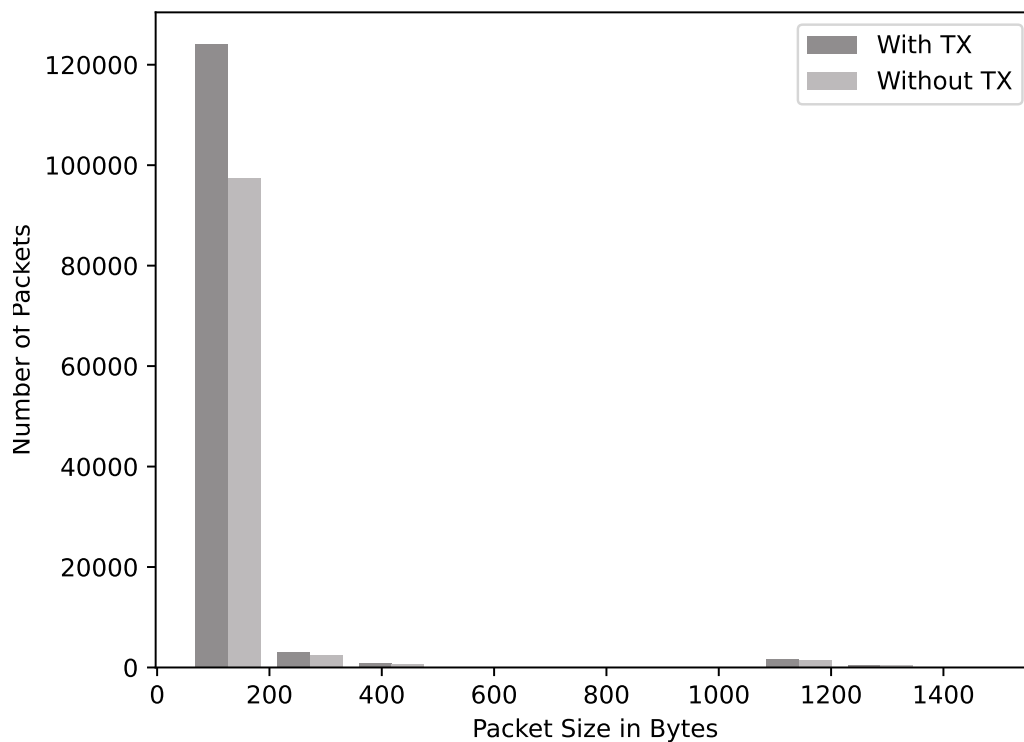


stack, which relies on TCP protocol, all of its traffic is encrypted. Therefore, it is not possible to distinguish packets belonging to different Ethereum devp2p communication processes by manually inspecting its content. This is also shown in the current available literature (GABA et al., 2022), corroborating the need of other techniques to classify Ethereum traffic. Thus, in this work, we have tried to take advantage of ML techniques to categorize Ethereum traffic and to identify which packets belonged to transactions performed in the BC.

### 5.2.2 Categorization and Evaluation of Transaction Packets

In order to attempt to categorize Ethereum transaction traffic, we have compared some of the differences in packets extracted from the executions of the proposed scenario. Figure 5.5 compares the amount of Ethereum packets and their size in bytes. Packets were obtained from PCAP files that resulted of the executions of the proposed scenario, containing Ethereum traffic with and without transactions, respectively.

Figure 5.5: Ethereum traffic analysis



Source: (The Author, 2024)

It is possible to perceive that transactions that occurred in the proposed scenario have increased the amount of packets in the range of 0 to 200 bytes. This goes in accordance with the logical structure of a transaction that is defined in Ethereum’s Whitepaper (BUTERIN, 2014), as depicted in Figure 5.6, that demonstrates the expected structure of a transaction packet in Ethereum BC.

Figure 5.6: Structure of a transaction message in Ethereum

<b>Nonce</b>	Up to 32 bytes
<b>GasPrice</b>	Up to 32 bytes
<b>GasLimit</b>	Up to 32 bytes
<b>Recipient</b>	20 bytes
<b>Value</b>	Up to 32 bytes
<b>Data</b>	24.576 bytes *
<b>V</b>	1 byte
<b>R</b>	32 bytes
<b>S</b>	32 bytes

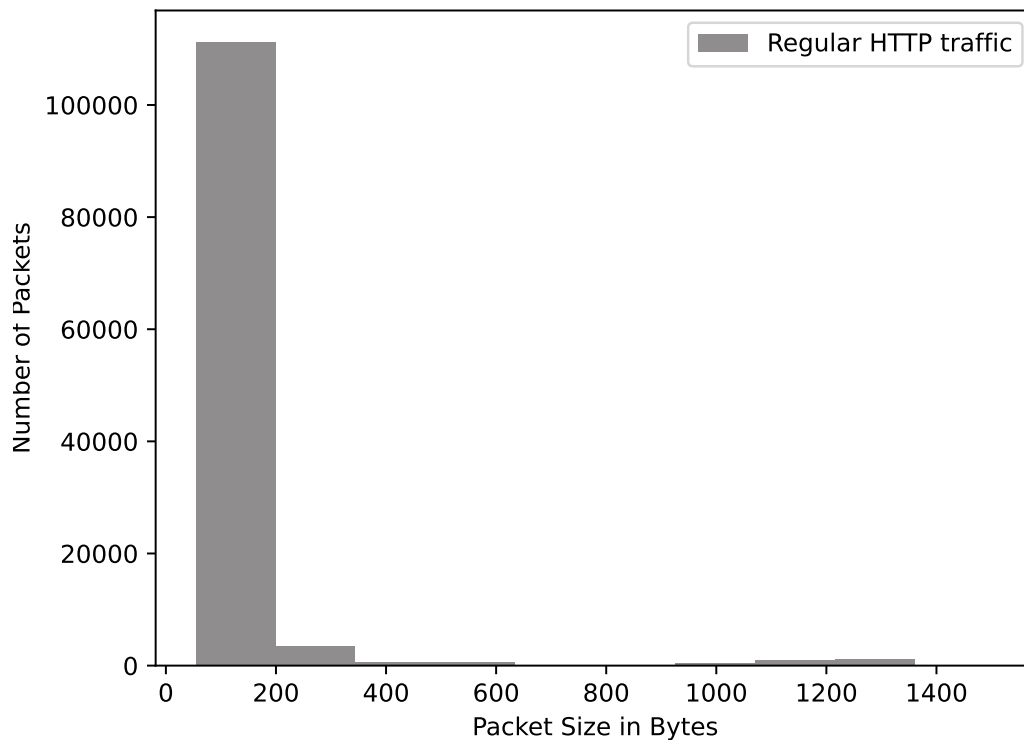
**Transaction Message Structure**

Source: (The Author, 2024)

Data is used in Ethereum to deploy or interact with smart contracts, as shown in Section 2.1.4. Ethereum Improvement Proposal (EIP) 170 (eips.ethereum.org, 2016) has introduced a limit to the size of Contract codes deployed in the BC to 24.576 bytes, to avoid high computational costs of reading and preprocessing code to the EVM. Hence, as transactions that were performed in the proposed scenario did not contain any data (*i.e.*, smart contracts deployment or any other additional data), it is expected that their sizes in bytes are up limited by the sum of the maximum possible size of other fields.

Although we can infer that transaction packets in the dataset are in the range of 100 to 200 bytes in size, we could not safely differentiate them from other devp2p Ethereum-related messages in the network that could coincide with the same size, nor with other regular HTTP traffic packets. Figure 5.7 demonstrates the variation of packet size in regular HTTP traffic generated with `noisy`, that simulates web browsing.

Figure 5.7: Variation of packet size in regular HTTP traffic



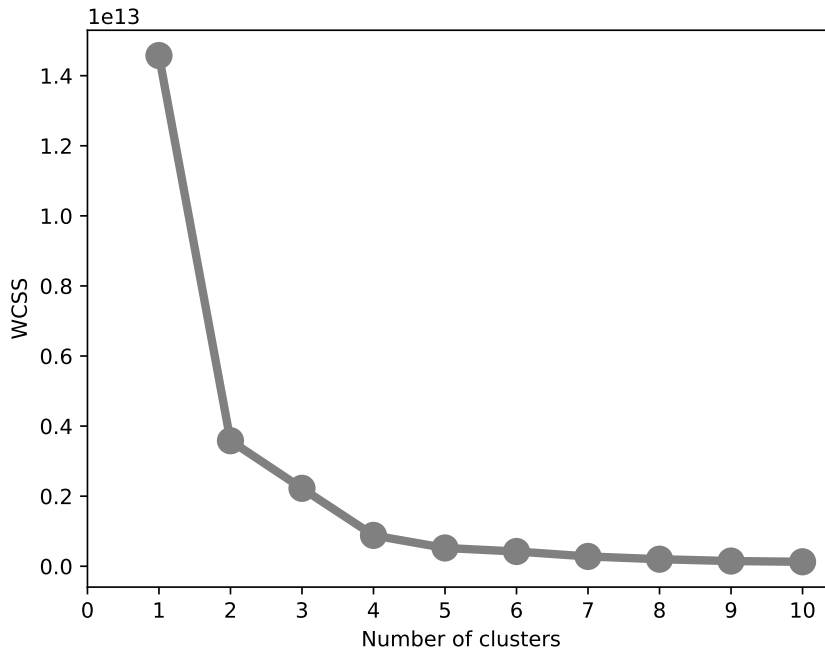
Source: (The Author, 2024)

Traffic generated from the simulation of web browsing displays a great amount of packets in the range of 50 to 200 bytes, coinciding with the inferred size of Ethereum transaction packets in the scenario. Further, we rely on the KMeans algorithm, which is an unsupervised ML algorithm, to clusterize Ethereum traffic and perceive if there were any possible packet size patterns that could be related to the type of message in the BC. KMeans was selected due to its ease of implementation and the advantages it offers regarding the visualization of large datasets, as the one obtained in this work. Clustering with KMeans could help to identify patterns and structures by grouping and plotting Ethereum packet data, aiding in the detection of relationships between attributes that might not be apparent when analyzing individual packets.

To identify the optimal number of clusters (*i.e.*, the  $k$  value), we rely on the Elbow method, a heuristic that is often used in KMeans algorithm to analyze the variation of Within-Cluster Sum of Squares (WCSS), which is the sum of the square distance between the centroid of a cluster and points that belong to it. The Elbow method varies  $k$  and fits the model, recording the WCSS for each iteration. By plotting the variation of WCSS with the increment of the number of clusters, it is possible to choose a value of  $k$  where

the rate of decrease of WCSS is at its maximum curvature, *i.e.*, the "*elbow*" of the curve, indicating that it is not worth increasing the number of clusters past this point. Figure 5.8 depicts the Elbow method using KMeans algorithm applied to Ethereum traffic obtained from the execution of the scenario proposed in this work.

Figure 5.8: Optimal number of cluster analysis using Elbow Method

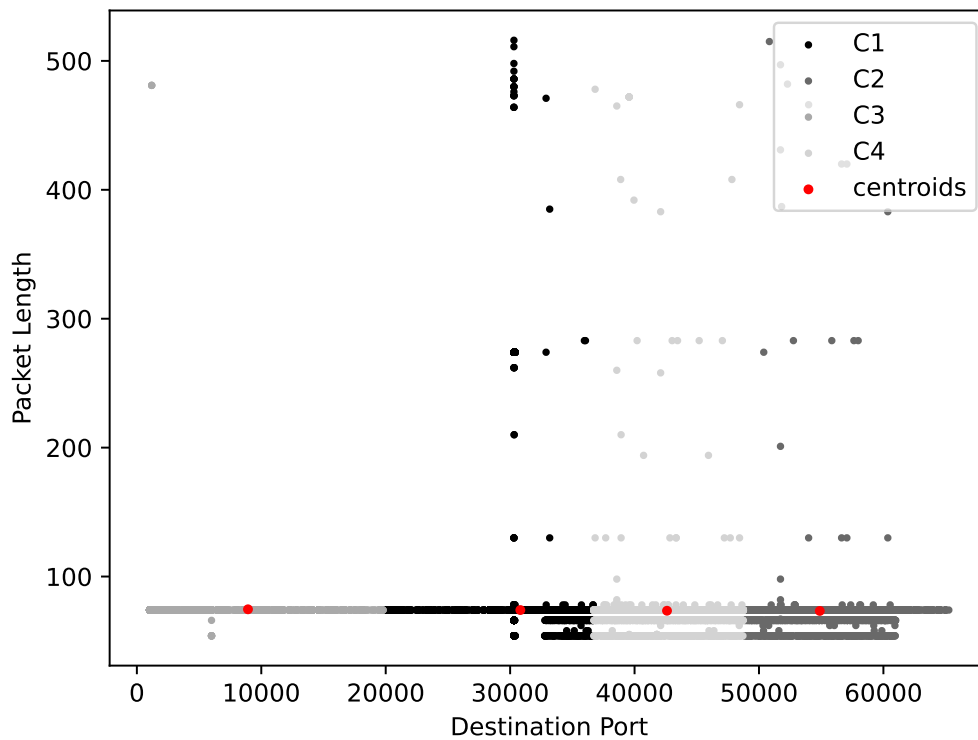


Source: (The Author, 2024)

From the results obtained by the use of the Elbow method, it is possible to perceive that the optimum number of clusters appears to be  $k=4$ . Hence, this is the value that is chosen to proceed for future analysis of Ethereum traffic. Figures 5.9 and 5.10 demonstrates the obtained results from applying the KMeans algorithm using  $k=4$  in Ethereum TCP and UDP traffic, respectively.

As transactions performed in the experiment are structured in the same way, *i.e.*, they do not have any smart contract interaction, they tend to present the same packet size, thus being perceived at the same place in the plotted clusterization of TCP traffic in Figure 5.9, in the region of 100 to 200 bytes close to the range of 30000 on the x-axis. Based on the results obtained from the clusterization using KMeans, it can be concluded that categorizing Ethereum traffic by packet size and destination port alone was not reasonable as there was no clear correlation between these features and Ethereum traffic type (*e.g.*, transactions, block synchronization, and node discovery). This behaviour might be due to the encryption of the traffic and the standardized transaction structure of Ethereum. Further, the experiments performed in this work did not include the presence of smart

Figure 5.9: KMeans Clusterization of Ethereum TCP Traffic



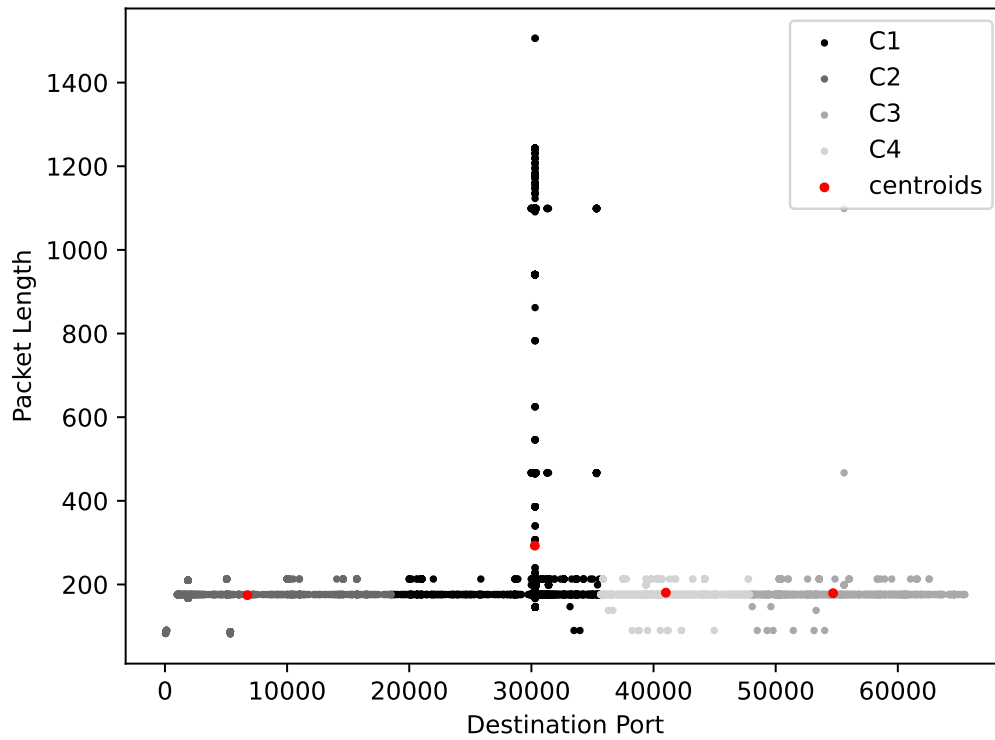
Source: (The Author, 2024)

contract interaction, which is planned to be investigated in future work.

### 5.3 Challenges and Remarks

One of the main challenges faced during this work was the fact that the packets under the `devp2p` protocol stack were encrypted using ECIES asymmetric encryption method, as mentioned in Subsection 2.1.1. Therefore, decryption of a packet's content to analyze it and classify the packet as BC traffic would require knowing the ephemeral key established during the handshake of two nodes, which is not attainable in a real-case scenario. The literature shows that the most common approach to overcome such challenge, when analyzing traffic of cryptojackers, is to adopt ML techniques to successfully identify and differentiate them from regular HTTP traffic (GABA et al., 2022). Though, as no available work had targeted Ethereum transaction traffic yet, we have tried to apply such techniques, *i.e.*, KMeans algorithm to clusterize Ethereum traffic and identify possible patterns.

Figure 5.10: KMeans Clusterization of Ethereum UDP Traffic



Source: (The Author, 2024)

Although the clusterization of Ethereum traffic did not lead to promising results when trying to identify possible patterns and comparing it to regular HTTP traffic, the generated novel dataset, as well as the remarks on the expected structure of Ethereum transactions and the replicable scenario using LST provided by this work, can foster further research on the use of ML techniques to identify Ethereum traffic and transactions in the BC in real-time using SDN.

## 6 SUMMARY, CONCLUSION AND FUTURE WORK

The identification of BC traffic has been a topic of increasing interest in recent years. The wide range of BC applications and the rise of its use in different areas (MON-RAT; SCHELÉN; ANDERSSON, 2019) leads to the development of different QoS requirements to be applied by network administrators. Hence, the possibility of identifying BC-related traffic in real-time to detect possibly infected devices with cryptojackers, or to prioritize its traffic through a SDN controller, would be greatly boosted by the automation of detection of BC packets in a network in real-time without the need of human supervision, thus reducing the costs to do so.

The use of SDN in the context of BC test scenarios has not been deeply explored in the available scientific literature yet, as shown in Chapter 3. The advantages offered by SDN through the separation of the control and data planes, *i.e.*, the customization of a virtual switch that can take action upon network flows or individual packets, can be beneficial to tackle the QoS requirements from BC applications. Though, the intrinsic variations of network experiments, that depends on several factors such as the underlying operating system, available resources and utilized softwares, are a barrier for the reproduction of test scenarios in network research. Most of the testbeds available are restricted to the context of the application that they were developed to, or are applicable with SDN (KAIHARA et al., 2022). Hence, in this work, we intended to provide an experiment that could serve as an example for reproducing other BC test scenarios using SDN with LST. The results obtained from the implementation of the network topology depicted in Figure 4.1 shows that the use of LST, as it is a highly configurable tool that offers an isolated environment for network experiments while leveraging SDN, allows an easy setup to conduct test scenarios with low effort. As it leverages Docker containers, any custom environment can be set up and deployed on the nodes, as shown in Listing 4.4. The use of a custom SDN controller deployed on the virtual switch container in LST also enhances the range of possible applications that can be researched in test scenarios.

As Ethereum cryptocurrency is the second largest by market capitalization (CoinMarketCap, 2024), it becomes an increasing target of malicious activity and its use in several areas is naturally expanded to different applications. Although there is some work available in the current scientific literature on identifying BC traffic using ML techniques, most of them are directed to other BCs such as Bitcoin, which is the most popular, or Monero, as its cryptocurrency is a popular in-browser cryptojacking target. Thus, this

work also aimed to tackle the topic on Ethereum BC. Analysis conducted on the extracted packets from Ethereum traffic demonstrates the typical expected structure on a transaction message, when there is no smart contract data attached to it. Even though there is a limit to the maximum size in bytes of deployed contract codes in transaction messages, and by consequence to the message itself, observations performed in Chapter 5 shows that relying on packet size alone is not sufficient to differ Ethereum traffic from regular HTTP traffic, as they often coincide in packet length. As the content of Ethereum messages transmitted through the `devp2p` stack protocols are encrypted, it is also not possible to rely on payload analysis to conduct classification of the BC packets. Thus, it is necessary to perform deeper inspection and combine different attributes to ML algorithms to successfully identify Ethereum traffic in real time. Further research could use LST to tackle the identification of Ethereum-related packets using ML techniques with and without smart contract data, as they are a common component in Ethereum transactions.

The employment of LST conducted in this work can serve as a base for future research on BC emulation and traffic analysis, as the scenario can be easily reproduced and the network topology can be extended up to the available resources on the host's machine. The novel dataset containing Ethereum and regular HTTP traffic provided by the executions of the proposed scenario in this work can be used in future work to explore the usage different ML algorithms, *e.g.*, SVM or CNN, to classify Ethereum traffic and transactions, as these algorithms are already used in other papers to identify several BCs traffic (NETO et al., 2020; KELTON et al., 2020). In addition, smart contract interactions can be emulated so that the related transactions can be included in the dataset.



## REFERENCES

- AZEVEDO, A. et al. Emulating an ethereum blockchain network using the Ist. In: **Anais da XX Escola Regional de Redes de Computadores**. Porto Alegre, RS, Brasil: SBC, 2023. p. 67–72. ISSN 0000-0000. Available from Internet: <<https://sol.sbc.org.br/index.php/errc/article/view/26007>>.
- bcsec.org. **Ethereum Wireshark Dissector**. 2018. <[https://github.com/bcsecorg/ethereum\\_devp2p\\_wireshark\\_dissector](https://github.com/bcsecorg/ethereum_devp2p_wireshark_dissector)>.
- BELOTTI, M. et al. A Vademecum on Blockchain Technologies: When, Which and How. **IEEE Access**, IEEE, p. 3796–3838, July 2019.
- BUTERIN, V. **Ethereum Whitepaper: A Next-Generation Smart Contract and Decentralized Application Platform**. 2014. <[https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum\\_Whitepaper\\_-\\_Buterin\\_2014.pdf](https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf)>.
- CABAJ, K.; GREGORCZYK, M.; MAZURCZYK, W. Software-defined networking-based crypto ransomware detection using http traffic characteristics. **Computers & Electrical Engineering**, v. 66, p. 353–368, 2018.
- CAPROLU, M. et al. Cryptomining Makes Noise: Detecting Cryptojacking via Machine Learning. **Computer Communications**, v. 171, p. 126–139, 2021.
- CoinMarketCap. **CoinMarketCap - Criptocurrency Market Capitalization**. 2024. <<https://coinmarketcap.com>>.
- DEV2P2P Community. **DEV2P2P - Ethereum Peer-to-Peer Networking Specifications**. 2023. <<https://github.com/ethereum/devp2p>>.
- eips.ethereum.org. **EIP-170: Contract code size limit**. 2016. <<https://eips.ethereum.org/EIPS/eip-170>>.
- ethereum.org. **Ethereum Networking Layer**. 2023. <<https://ethereum.org/en/developers/docs/networking-layer>>.
- FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The Road to SDN: An Intellectual History of Programmable Networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 2, p. 87–98, apr 2014. ISSN 0146-4833.
- GABA, S. et al. Machine learning for detecting security attacks on blockchain using software defined networking. In: **IEEE International Conference on Communications Workshops (ICC)**. Seoul, KOR: [s.n.], 2022.
- geth.ethereum.org. **Geth - Official Go implementation of the Ethereum protocol**. 2024. <<https://geth.ethereum.org/>>.
- GILL, S.; LEE, B.; QIAO, Y. Containerchain: A Blockchain System Emulator based on Mininet and Containers. In: **IEEE International Conference on Blockchain (Blockchain 2021)**. Melbourne, Australia: [s.n.], 2021. p. 1–7.

HURY, I. **Noisy - Random HTTP/DNS Traffic Generator**. 2018. <<https://github.com/1tayH/noisy>>.

KAIHARA, A. et al. LST: Testbed Emulado Leve para Redes SDN Aplicado ao Contexto de Segurança. In: **Anais Estendidos do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Fortaleza, Ceará: [s.n.], 2022. p. 41–48. ISSN 2177-9384.

KELTON, C. et al. Browser-based deep behavioral detection of web cryptomining with coinspy. In: **Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)**. San Diego, CA, USA: [s.n.], 2020.

KIM, J. et al. Anomaly detection based on traffic monitoring for secure blockchain networking. In: **IEEE International Conference on Blockchain and Cryptocurrency (ICBC)**. [S.l.: s.n.], 2021.

KREUTZ, D. et al. Software-Defined Networking: A Comprehensive Survey. **Proceedings of the IEEE**, IEEE, v. 103, p. 14–76, December 2014.

MCKEOWN, N. et al. OpenFlow: Enabling Innovation in Campus Networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 2, p. 69–74, mar 2008. ISSN 0146-4833.

MONRAT, A. A.; SCHELÉN, O.; ANDERSSON, K. A Survey of Blockchain From the Perspectives of Applications, Challenges, and Opportunities. **IEEE Access**, v. 7, p. 117134–117151, August 2019.

MUÑOZ, J. Z. i. Detection of Bitcoin Miners From Network Measurements. 2019.

MUÑOZ, J. Z. i; SUÁREZ-VARELA, J.; BARLET-ROS, P. Detecting cryptocurrency miners with netflow/ipfix network measurements. In: **IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)**. Wuhan, CH: [s.n.], 2021.

NAKAMOTO, S. **Bitcoin: A Peer-to-Peer Electronic Cash System**. 2009. <<https://bitcoin.org/bitcoin.pdf>>.

NASEEM, F. et al. Minos\*: A lightweight real-time cryptojacking detection system. In: **Network and Distributed Systems Security (NDSS) Symposium**. [S.l.: s.n.], 2021.

NETO, H. N. C. et al. MineCap: Super Incremental Learning for Detecting and Blocking Cryptocurrency Mining on Software-Defined Networking. **Annals of Telecommunications**, v. 75, p. 121–131, 2020.

NING, R. et al. Capjack: Capture in-browser crypto-jacking by deep capsule network through behavioral analysis. In: **IEEE INFOCOM 2019 - IEEE Conference on Computer Communications**. Paris, FR: [s.n.], 2019.

Nippon Telegraph and Telephone Corporation. **PCAP file library**. 2014. <[https://ryu.readthedocs.io/en/latest/library\\_pcap.html](https://ryu.readthedocs.io/en/latest/library_pcap.html)>.

PAULAVIČIUS, R.; GRIGAITIS, S.; FILATOVAS, E. An Overview and Current Status of Blockchain Simulators. **IEEE International Conference on Blockchain and Cryptocurrency (ICBC)**, IEEE, Jun 2021.

RODRIGUEZ, J. D. P.; POSSEGA, J. Rapid: Resource and api-based detection against in-browser miners. In: **Annual Computer Security Applications Conference (ACSAC)**. San Juan, PR: [s.n.], 2018.

RUSSO, M.; SRNDIĆ, N.; LASKOV, P. Detection of illicit cryptomining using network metadata. **EURASIP Journal on Information Security**, v. 1, p. 1–20, 2021.

Ryu SDN Framework Community. **Ryu SDN Framework**. 2017. <<https://ryu-sdn.org/>>.

SANKA, A. I.; CHEUNG, R. C. A Systematic Review of Blockchain Scalability: Issues, Solutions, Analysis and Future Research. **Journal of Network and Computer Applications**, v. 195, p. 103232, 2021.

Scapy Community. **Scapy Packet Manipulation Library**. 2024. <<https://scapy.net>>.

SCHEID, E. J. et al. Blockchains and Distributed Ledgers Uncovered: Clarifications, Achievements, and Open Issues. In: **Advancing Research in Information and Communication Technology**. Cham, Switzerland: Springer, 2021, (IFIP AICT Festschrifts, v. 600). p. 289–317.

TEKINER, E.; ACAR, A.; ULUAGAC, A. S. A lightweight iot cryptojacking detection mechanism in heterogeneous smart home networks. In: **Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)**. San Diego, CA, USA: [s.n.], 2022.

VUJIČIĆ, D.; JAGODIĆ, D.; RANĐIĆ, S. Blockchain technology, bitcoin, and ethereum: A brief overview. In: **2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)**. [S.l.: s.n.], 2018.

XIA, W. et al. A Survey on Software-Defined Networking. **IEEE Communications Surveys & Tutorials**, IEEE, v. 17, p. 27–51, June 2014.

All links accessed on 06/02/2024

## APPENDIX A — PUBLISHED PAPER – ERRC 2023

AZEVEDO, Andrei C.; SCHEID, Eder J.; FRANCO, Muriel F.; GRANVILLE, Lisandro Z.. Emulating an Ethereum Blockchain Network Using the LST. In: ESCOLA REGIONAL DE REDES DE COMPUTADORES (ERRC), 20. , 2023, Porto Alegre/RS. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2023 . p. 67-72. DOI: <https://doi.org/10.5753/errc.2023.918>.

- **Title:** *Emulating an Ethereum Blockchain Network Using the LST*
- **Contribution:** An emulated scenario of a real BC network.
- **Abstract:** Besides the main blockchain use-case of exchanging cryptocurrencies, Distributed Applications (DApps) can also be developed on top of such a technology. However, due to the size of popular blockchains and price, testing these DApps in a real-world environment becomes challenging. Thus, blockchain emulators were proposed to address such as issue. This paper presents the experience of emulating an Ethereum network using a Docker-based lightweight testbed developed for Software Defined Networks (SDN).
- **Status:** Published
- **Qualis:** -
- **Conference:** 20ª Escola Regional de Redes de Computadores (ERRC 2023)
- **Date:** October 23 - October 25, 2023
- **Local:** Porto Alegre, RS, Brasil
- **URL:** <<https://sol.sbc.org.br/index.php/errc/article/view/26007>>
- **Digital Object Identifier (DOI):** <<https://doi.org/10.5753/errc.2023.918>>

# Emulating an Ethereum Blockchain Network Using the LST

Andrei C. Azevedo, Eder J. Scheid, Muriel F. Franco, Lisandro Z. Granville

Informatics Institute (INF) – Federal University of Rio Grande do Sul (UFRGS)  
Porto Alegre – RS – Brazil

{acazevedo,ejscheid,mffranco,granville}@inf.ufrgs.br

**Abstract.** *Besides the main blockchain use-case of exchanging cryptocurrencies, Distributed Applications (DApps) can also be developed on top of such a technology. However, due to the size of popular blockchains and price, testing these DApps in a real-world environment becomes challenging. Thus, blockchain emulators were proposed to address such as issue. This paper presents the experience of emulating an Ethereum network using a Docker-based lightweight testbed developed for Software Defined Networks (SDN).*

## 1. Introduction

Blockchain is a technology based on the concept of distributed ledgers that stores information in a decentralized and distributed network [Scheid et al. 2021]. It was first proposed in 2008, as the database and means to solve the double-spending problem for the Bitcoin cryptocurrency [Nakamoto 2009], and since then, its adoption has expanded to several other areas such as finances, cybersecurity and Internet of Things (IoT) [Monrat et al. 2019], as well as to other emerging cryptocurrencies, such as Ethereum [Buterin 2014].

Since the architecture of blockchains is complex, comprising several different layers (*e.g.*, network layer, data model layer, execution layer and application layer) [Belotti et al. 2019], it is difficult to create and maintain an actual blockchain environment for evaluation purposes. Further, the size of blockchains, such as Bitcoin and Ethereum, is over 400 GB, requiring dedicated hardware [Sanka and Cheung 2021]. Fortunately, it is possible to resort to blockchain emulation and simulators to avoid such difficulties. Simulation aims to reproduce a blockchain system model, enabling its evaluation in a parameterized way, without the need to implement the entire system [Paulavičius et al. 2021] whereas emulation aims to replicate the behavior of a system as closely as possible [Gill et al. 2021].

Thus, in this work, we rely on the Lightweight SDN Testbed (LST) [Kaihara et al. 2022] to emulate an Ethereum-based blockchain network containing different nodes. We defined a scenario with one signer creating and attesting blocks and two regular nodes sending transactions. We show that it is possible to emulate an Ethereum network in an flexible and reproducible manner while being possible to send transactions among the nodes and to synchronize the blockchain, enabling a complete evaluation of the system relying on real-world blockchain software.

The remainder of this paper is organized as follows. Section 2 presents an overview of LST and the Ethereum blockchain. Section 3 discusses related work on Ethereum blockchain simulator and emulators available in the literature. Section 4 describes the scenario for running the experiment, while Section 5 elaborates on its results. Finally, Section 6 concludes this paper and indicates future work on the topic.

## 2. Background

This section describes the two main technologies employed in this paper, LST and the Ethereum blockchain.

### 2.1. Lightweight SDN Testbed (LST)

LST [Kaihara et al. 2022] is a lightweight and easy-to-use tool for SDN and security studies. It allows reproducing scenarios in the context of SDN through the virtualization of physical infrastructures, by taking advantage of Docker containers. Through the virtualization offered by Docker containers, it is possible to achieve a highly configurable and isolated environment.

The tool relies on the Ryu Controller, a SDN framework for network management and control applications that supports several protocols, such as OpenFlow. Thus, it allows to perform different actions on the virtual switches *e.g.*, network analysis, changing network route policies and identifying malicious attacks. Further, as it relies on Docker containers, such a controller can be replaced and modified in a flexible manner.

### 2.2. Ethereum

Ethereum was proposed in 2015 as a blockchain able to support the creation of versatile applications [Buterin 2014]. It offers, within its blockchain ecosystem, a way for developers to create Decentralized Applications (DApps) using a Turing-complete programming language. With such a language, developers define immutable blockchain-based smart contracts that contain specific rules for the enforcement of transactions used by DApps. Thus, by providing this language, Ethereum became the *De Facto* standard for DApps.

Unlike Bitcoin, where the blockchain state is defined by Unspent Transaction Output (UTXO) transactions, Ethereum uses account-based states, where each account is defined by a set of fields (*e.g.*, balance), and state transitions are transactions of information or value between different accounts. Ethereum relies on the Proof-of-Stake (PoS) consensus method, where a node must stake capital (*i.e.*, Ethereum coins - ETH), into a smart contract. Only nodes with stake in the smart contract can propose and validate blocks; if the node does not follow a set of rules or propose invalid blocks, its stake is reduced, decreasing its chance of proposing blocks.

## 3. Related Work

There are several blockchain simulators available in the literature [Paulavičius et al. 2021]. However, in terms of blockchain emulation, there are few approaches in the literature. Thus, as the focus of this paper is on emulation, we only describe Ethereum-based emulators.

[Gill et al. 2021] emulates a real Ethereum blockchain by leveraging Container-net [Peuster et al. 2018], a Mininet fork that allows using Docker containers as hosts. It also provides network monitoring feature for retrieving statistics through a Netflow collector, enabling traffic analysis. Although Mininet does support OpenFlow protocol, the proposed blockchain system emulator does not provide an interface for setting up a SDN Controller as does LST. Thus, although traffic can be analyzed with Netflow collected statistics, it is harder to perform any actions on incoming packets in the emulated system.

[Wang et al. 2019] presents an emulator that is highly scalable, but is designed to support only public blockchains that are based on the Proof-of-Work (PoW) consensus protocol. In contrast to LST, which uses Docker containers for each node, the proposed emulator is a Java application; thus, having a less isolated and flexible environment. In another work, [Polge et al. 2021] only emulates the networking layer of the blockchain, the remainder of the layers is simulated. Hence, it cannot be considered a full-fledged blockchain emulator but rather a hybrid emulator.

Based on such a review, it can be seen that, in the literature, one only approach focuses on using containers to emulate a blockchain network. However, SDN and lightweight emulation using Docker is not explored in this context. Hence, using LST as a flexible blockchain emulator with SDN to create a Ethereum network or any blockchain network presents an interesting opportunity.

#### 4. Scenario

Figure 1 illustrates the blockchain network scenario emulated using LST. The host provides the required CPU, RAM, Disk for the virtualization of the elements of LST. Two blockchain nodes and a blockchain signer node are initialized as docker containers, and later connected to the virtual Switch container, creating the blockchain network topology. The Switch is connected to a custom Ryu Controller container, that enables us to record and later inspect the network flows. It also uses the network interface provided by the host, thus allowing the blockchain emulated system to have Internet connection.

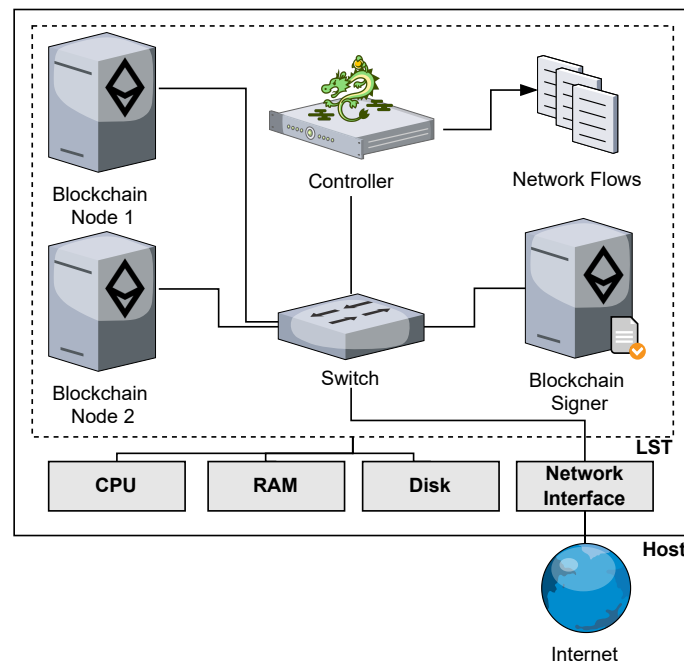


Figure 1. Emulated Scenario with LST

In this scenario, we have created a private Ethereum blockchain that relies on the Proof-of-Authority (PoA) consensus protocol, where one or more nodes are assigned as validators for the entire network. We define one of the participating nodes as the Ethereum Block Signer. This node contains the genesis file for our private Ethereum



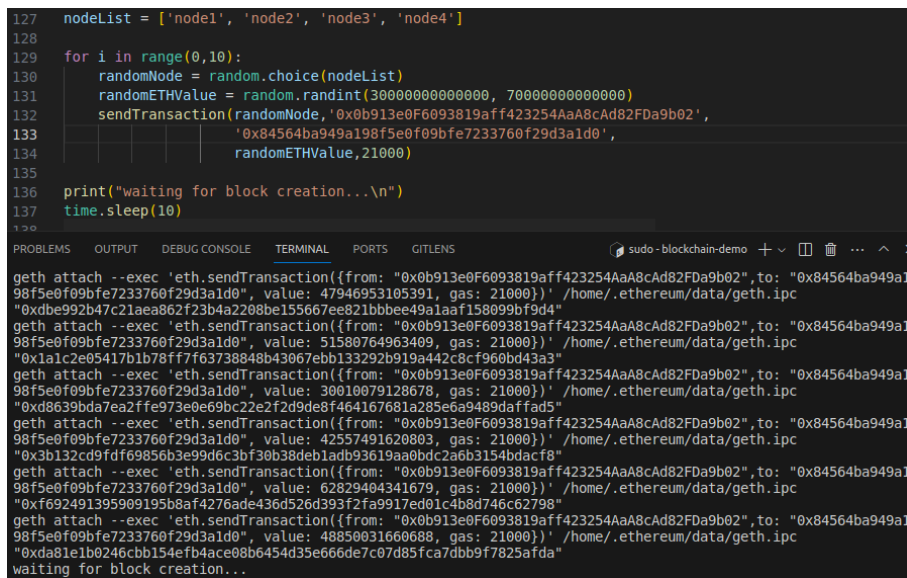


We can also see that LST is a very flexible tool that allows us to easily set up new nodes in the blockchain network and submit transactions with different ETH values. Figure 3 depicts an example of a different topology containing four common nodes (two more nodes than the original scenario), and randomly sending several transactions with different ETH values among them (see line 132). This allows us to verify the communication between several nodes in the network with the signer and verify if all the nodes are synchronizing with the blockchain. Further, with the use of LST and the experiment defined as a Python script, it can be repeated by other researchers to achieve the same result, fostering reproducible research.

```

127 nodeList = ['node1', 'node2', 'node3', 'node4']
128
129 for i in range(0,10):
130     randomNode = random.choice(nodeList)
131     randomETHValue = random.randint(3000000000000, 7000000000000)
132     sendTransaction(randomNode, '0x0b913e0f6093819aff423254AaA8cAd82Fda9b02',
133                     '0x84564ba949a198f5e0f09bfe7233760f29d3a1d0',
134                     randomETHValue, 21000)
135
136 print("waiting for block creation...\n")
137 time.sleep(10)
138
139

```



```

geth attach --exec 'eth.sendTransaction({from: "0x0b913e0f6093819aff423254AaA8cAd82Fda9b02", to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 47946953105391, gas: 21000})' /home/.ethereum/data/geth.ipc
"0xd8be992b47c21aea862f23b4a2208be155667ee821bbbee49a1aaf158099bf9d4"
geth attach --exec 'eth.sendTransaction({from: "0x0b913e0f6093819aff423254AaA8cAd82Fda9b02", to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 51580764963409, gas: 21000})' /home/.ethereum/data/geth.ipc
"0x1a1c2e05417b1b78ff7f63738848b43067eb133292b919a442c8cf960bd43a3"
geth attach --exec 'eth.sendTransaction({from: "0x0b913e0f6093819aff423254AaA8cAd82Fda9b02", to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 30010979128678, gas: 21000})' /home/.ethereum/data/geth.ipc
"0xd8639bda7ea2ffe973e0e69bc22e2f2d9de8f464167681a285e6a9489dafad5"
geth attach --exec 'eth.sendTransaction({from: "0x0b913e0f6093819aff423254AaA8cAd82Fda9b02", to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 42557491620803, gas: 21000})' /home/.ethereum/data/geth.ipc
"0x3b132cd9fd96856b3e99d6c3bf30b38deb1adb93619aa0bdc2a6b3154bdac8"
geth attach --exec 'eth.sendTransaction({from: "0x0b913e0f6093819aff423254AaA8cAd82Fda9b02", to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 62829404341679, gas: 21000})' /home/.ethereum/data/geth.ipc
"0xf692491395909195b8af4276ade436d526d393f2fa9917ed01c4b8d746c62798"
geth attach --exec 'eth.sendTransaction({from: "0x0b913e0f6093819aff423254AaA8cAd82Fda9b02", to: "0x84564ba949a198f5e0f09bfe7233760f29d3a1d0", value: 48850031660688, gas: 21000})' /home/.ethereum/data/geth.ipc
"0xda81e1b0246cb154efb4ace08b6454d35e666de7c07d85fca7dbb9f7825afda"
waiting for block creation...

```

**Figure 3. Sending several transactions from any of the four participating nodes with different ETH values.**

## 6. Conclusion and Future Work

In this paper, we have demonstrated that it is feasible to emulate an Ethereum blockchain using Docker containers and SDN with the use of LST, which is a versatile and highly configurable tool. Thus, allowing us to perform several analyses and actions on the emulated network. By leveraging Docker container technology, the tool creates an isolated environment that is ideal for testing purposes on blockchain systems without relying on a testnet, which might require prohibitive hardware resources and human effort for configuration of nodes.

Our experiments demonstrated that setting up new nodes and sending transactions with different values in the emulated Ethereum blockchain network using LST is a straightforward task. In addition, the experiments can be repeated with predictable results by other researchers aiming to test novel blockchain-based approaches and DApps in a real-world-like blockchain network with minimal effort.

For future work, it could be investigated the use of the SDN controller to identify Ethereum packets in the network traffic so that cryptojacker traffic is mitigated and to create smart contract interactions simulations with several real-world Ethereum nodes and traffic conditions.

## Acknowledgment

This work was supported by The São Paulo Research Foundation (FAPESP) under the grant number 2020/05152-7, the PROFISSA project.

## References

- Belotti, M., Božic, N., Pujolle, G., and Secci, S. (2019). A Vademecum on Blockchain Technologies: When, Which and How. *IEEE Access*, pages 3796–3838.
- Buterin, V. (2014). Ethereum Whitepaper: A Next-Generation Smart Contract and Decentralized Application Platform. [https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum\\_Whitepaper\\_-\\_Buterin\\_2014.pdf](https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf).
- Gill, S., Lee, B., and Qiao, Y. (2021). Containerchain: A Blockchain System Emulator based on Mininet and Containers. In *IEEE International Conference on Blockchain (Blockchain 2021)*, pages 1–7, Melbourne, Australia.
- Kaihara, A., Bondan, L., Gondim, J., Rodrigues, G., Marotta, M., and Rodrigues, G. (2022). LST: Testbed Emulado Leve para Redes SDN Aplicado ao Contexto de Segurança. In *Anais Estendidos do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 41–48, Fortaleza/CE.
- Monrat, A. A., Schelén, O., and Andersson, K. (2019). A Survey of Blockchain From the Perspectives of Applications, Challenges, and Opportunities. *IEEE Access*, 7:117134–117151.
- Nakamoto, S. (2009). Bitcoin: A Peer-to-Peer Electronic Cash System. <https://bitcoin.org/bitcoin.pdf>.
- Paulavičius, R., Grigaitis, S., and Filatovas, E. (2021). An Overview and Current Status of Blockchain Simulators. *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*.
- Peuster, M., Kampmeyer, J., and Karl, H. (2018). Containernet 2.0: A Rapid Prototyping Platform for Hybrid Service Function Chains. In *IEEE Conference on Network Softwarization and Workshops (NetSoft 2018)*, pages 335–337, Montreal, Canada.
- Polge, J., Ghatpande, S., Kubler, S., Robert, J., and Le Traon, Y. (2021). BlockPerf: A Hybrid Blockchain Emulator/Simulator Framework. *IEEE Access*, 9:107858–107872.
- Sanka, A. I. and Cheung, R. C. (2021). A Systematic Review of Blockchain Scalability: Issues, Solutions, Analysis and Future Research. *Journal of Network and Computer Applications*, 195:103232.
- Scheid, E. J., Rodrigues, B., Killer, C., Franco, M., Rafati, S., and Stiller, B. (2021). Blockchains and Distributed Ledgers Uncovered: Clarifications, Achievements, and Open Issues. In *Advancing Research in Information and Communication Technology*, volume 600 of *IFIP AICT Festschriften*, pages 289–317. Springer, Cham, Switzerland.
- Wang, X., Al-Mamun, A., Yan, F., and Zhao, D. (2019). Toward Accurate and Efficient Emulation of Public Blockchains in the Cloud. In *International Conference on Cloud Computing (CLOUD 2019)*, pages 67–82, San Diego, CA, USA. Springer International Publishing.