END-OF-DEGREE PROJECT

TITULO: Decentralized collaborative system for the creation and validation of a Blacklist using blockchain

AUTOR/AUTHOR: Rafael Malla Martinez

DEGREE: Bachelor's Degree in Telematic Engineering

TUTOR/A: Jesús Rodríguez Molina

DEPARTMENT: Telematics Engineering and Electronics

VºBº TUTOR/A

Members of the Qualifying Tribunal:

PRESIDENT: Jerónimo López-Salazar

TUTOR: Jesús Rodríguez Molina

SECRETARY: Pedro castillejo Parrilla

Date of reading:

Qualification:

June 2025

Acknowledgements

To Jesus, my tutor, for his guidance and help.

A mi familia por su apoyo.

A Alejandra por quererme, aunque se lo pusiera difícil.

Resumen

Este proyecto presenta el diseño e implementación de un sistema distribuido para la detección y prevención de ataques de denegación de servicio (DoS, *Denial of Service*). La solución propuesta emplea tecnología blockchain con el fin de almacenar de manera segura metadatos asociados a flujos de tráfico malicioso. El sistema está compuesto por múltiples nodos que operan de forma autónoma.

Integra un módulo de monitorización capaz de identificar patrones anómalos mediante un algoritmo de aprendizaje supervisado adaptable (*supervised learning algorithm*). Cuando un flujo de red es clasificado como malicioso, su información es almacenada en un contrato inteligente desplegado en la red de pruebas de VeChain, lo que garantiza la inmutabilidad y persistencia de los registros.

Inicialmente, se utilizó Hyperledger Fabric como plataforma blockchain. No obstante, tras un proceso de evaluación experimental detallado en esta memoria, se procedió a migrar la infraestructura a la red VeChain. Esta decisión mejoró tanto la escalabilidad como la robustez del sistema.

La arquitectura desarrollada está compuesta por dos nodos maestros y un nodo auxiliar. El primer nodo se encarga de la captura y análisis de tráfico; el segundo gestiona la interacción con la red blockchain y mantiene la lista negra de flujos maliciosos. El nodo auxiliar tiene como objetivo realizar pruebas de validación del sistema, generando tráfico tanto legítimo como malicioso hacia un sistema objetivo.

El sistema es capaz de analizar los flujos capturados, extraer sus características principales y clasificarlos con alta precisión. En el caso de que un flujo sea identificado como malicioso, se generan alertas parametrizadas en función de sus *hiperparámetros* de red. Cuando dichas alertas superan un umbral predefinido, la información relevante se registra automáticamente en la blockchain.

Adicionalmente, se implementa un servidor que actúa como interfaz de usuario (*frontend*), permitiendo gestionar la lista negra de forma visual. Este servidor mantiene una copia local de la lista negra registrada en la blockchain, lo que permite validar la descentralización y eficiencia de la solución propuesta.

En definitiva, este trabajo demuestra la viabilidad de integrar técnicas de ciberseguridad, sistemas distribuidos y tecnología blockchain para construir infraestructuras transparentes, resilientes y descentralizadas, orientadas a reforzar la confianza y la seguridad operativa de los sistemas de red.

June 2025

Abstract

This project presents the design and implementation of a distributed system for the detection and prevention of denial of service (DoS) attacks. The proposed solution uses blockchain technology to securely store metadata associated with malicious traffic flows. The system is made up of multiple nodes that operate autonomously.

It integrates a monitoring module capable of identifying anomalous patterns using an adaptive supervised learning algorithm. When a network flow is classified as malicious, its information is stored in a smart contract deployed on the VeChain network of tests, which guarantees the immutability and persistence of records.

Initially, Hyperledger Fabric was used as a blockchain platform. However, after an experimental evaluation process detailed in this memory, the infrastructure was migrated to the VeChain network. This decision improved both the scalability and robustness of the system.

The developed architecture is composed of two master nodes and one auxiliary node. The first node is responsible for traffic capture and analysis; The second manages the interaction with the red blockchain and maintains the blacklist of malicious flows. The auxiliary node has the objective of carrying out system validation tests, generating both legitimate and malicious traffic to an objective system.

The system is capable of analysing captured flows, extracting their main characteristics and classifying them with high precision. In the event that a stream is identified as malicious, parameterized alerts are generated based on its network hyperparameters. When alerts pass a predefined threshold, the relevant information is automatically registered on the blockchain.

Additionally, a server is implemented that acts as a user interface (frontend), allowing the blacklist to be managed visually. This server maintains a local copy of the blacklist registered on the blockchain, which allows validating the decentralization and efficiency of the proposed solution.

Ultimately, this work demonstrates the feasibility of integrating cybersecurity techniques, distributed systems and blockchain technology to build transparent, resilient and decentralized infrastructures, aimed at strengthening the trust and operational security of network systems.

Table of Contents

Acknowledgements 1
Resumen 3
Abstract 5
Table of Contents 7
Table of Illustrations 11
Index of tables 13
Acronyms 15
1 Introduction 17
1.1 Contributions 17
1.2 Project Structure 17
2 Technological Framework 19
2.1 Overview of the Utilized Technology 19
2.2 Introduction to Blockchain 20
2.2.1 Blockchain Nodes 21
2.2.2 Blocks in a Blockchain 22
2.2.3 Types of Blockchain 23
2.2.4 Consensus Mechanisms 23
2.2.5 Transactions and Transaction Process 24
2.2.6 Smart Contracts 24
2.3 HyperLedger Fabric 25
2.3.1 Network 25
2.3.2 Configtxgen 25
2.3.3 Protobuf 26
2.3.4 Channel Policies 26
2.3.5 Nodes 26
2.3.6 MSP & Identity Management 27
2.3.7 Channels 28
2.3.8 Transactions 29
2.3.9 Chaincode 29
2.3.10 HLF Consensus Mechanism 30
2.3.11 Ledger & Data Storage 31
2.3.12 Docker & Docker-Compose 31
2.4 VeChain 31
2.4.1 Consensus Algorithm 31
2.4.2 Token System 32

2.4.3 Tools & SDK 32

2.4.4 Use Cases 33

2.5 Denial of Service 34

2.5.1 Dos 34

2.5.2 DDoS 34

2.5.3 Variants 35

2.5.4 Attack Classification 37

2.5.5 Key Components 37

2.5.6 Mitigation Metrics 38

2.6 Modern Security Approach 38

2.6.1 Load Balancing 39

2.6.2 Rate Limiting 39

2.6.3 Network-level Filtering. 39

2.6.4 Traffic Filtering 39

2.6.5 Intrusion Prevention IPS 39

2.6.6 Machine Learning Based Techniques 39

2.6.7 Deep Learning & Neural Networks 40

2.6.8 Distributed Systems 40

2.6.9 SDN 40

2.7 Challenges 40

2.7.1 Reliability and traceability in Blacklist management 40

2.7.2 Entity collaboration 41

2.7.3 Automated integration between detection and logging 41

2.7.4 Data management and preprocessing for model training 41

3 Analysis, design, and solution implementation 43

3.1 Requirements 43

3.1.1 Functional Requirements 43

3.1.2 Non- Functional Requirements 43

3.2 Design 44

3.2.1 General Architecture 44

3.2.2 Blockchain Selection 45

3.2.3 Machine Learning Models Overview 47

3.2.4 Component Design 48

3.2.5 Object Oriented Design 49

3.2.6 Operational Flow 51

3.2.7 Data Model 54

3.2.8 Deployment Scheme 55

3.3 Solution Implementation 56

3.3.1 Development of the DoSDetector module 56

3.3.2 Implementing Persistence in Blockchain 58

3.3.3 Development of Blacklist Server module 59

3.3.4 Development of auxiliary traffic simulation module 61

3.3.5 General System Integration 63

4 Testing 65

4.1 Parameters Evaluation 65

4.2 Results of Statistical Analysis 68

5 Findings 71

5.1 Key Takeaways 71

5.2 Future Works 72

6 Cost Estimate 75

6.1 Tool Kit Costs 75

6.2 Personnel Expenses 75

7 Bibliography 77

Appendix I –Machine Learning Model Training and Evaluation Process 81

I.1 Reference dataset (CICIDS2017) 81

I.2 Utilized dataset (DBDoS2025) 81

I.3 Pre-processing 82

I.4 Model Comparison 82

Appendix II – Smart Contract in VeChain 87

II.1 Functions and Events Description 87

Appendix III –Hyperledger Fabric Chaincode 89

Appendix IV – Execution Manual 91

IV.1 System requirements 91

IV.2 Preparation 91

IV.3 System Execution 92

IV.4 Expected results 92

IV.5 Common Errors & Solutions 92

IV.6 Final Considerations 93

Table of Illustrations

Figure 1. Structure of a Block in a Blockchain 20
Figure 2. Workflow of Hyperledger Fabric [6] 21
Figure 3.Membership Service Provider (MSP) - HLF [30] 27
Figure 4. Attribute modification for Org 1 28
Figure 5. Entity register with personalized attributes 28
Figure 6. SYN Flood illustration [66] 36
Figure 7. General Architecture Scheme 45
Figure 8. Blockchain Technical Selection 46
Figure 9. KNN Model statistical recollection 47
Figure 10. DoSDetector Operational Flow 52
Figure 11. Blacklist Server Operational Flow 53
Figure 12. Malicious Traffic & System Flow 54
Figure 13. System modular layout 56
Figure 14. Algorithm loading snippet. 57
Figure 15. Logging snippet 57
Figure 16. DoS node Detection Flow 57
Figure 17. Class Structure in DoS Detection module 58
Figure 18. LogAttack.cjs Section 59
Figure 19. Synchronization & HTTP Response Flow 60
Figure 20. Controlled Shutdown Flow 61
Figure 21. Client SYNFlood code Snippets 62
Figure 22. Traffic flow 63
Figure 23. Model Stats Comparison DBDoS2025 65
Figure 24. Confusion Matrix. KNN Model 66
Figure 25. Statistical Time Comparison 67
Figure 26. PCA Scatter Plot of Labelled Traffic Data 69
Figure 27. SMOTE for class balancing 82
Figure 28. SVM parameter training saving 83
Figure 29. CICIDS2017 Metrics comparison, SVM omitted 84
Figure 30. F1-Score DBDoS2025 Dataset 85
Figure 31. HLF JSON Type 89
Figure 32. HLF Contract invoke 89
Figure 33. Model Training 91

Index of tables

Table 1. Acronym Table 15
Table 2. Comparison. HLF vs VeChain 46
Table 3. Time Statistics of all Models 66
Table 4. logAttack Stats 68
Table 5. Mean Classification Metrics per Class across Models 68
Table 6. Standard Deviation of Classification Metrics per Class........ 69
Table 7.Estimated HW & SW Costs........ 75
Table 8. Personnel Expenses 76
Table 9. Summarized CICIDS2017 models' stats comparison 84
Table 10. DBDoS2025 models stats........ 84
Table 11. VeChain SC Functions 87
Table 12. VeChain SC Events........ 88
Table 13. HLF Chaincode Functions........ 89

Acronyms

Table 1. Acronym Table

Acronym	Description
MSP	Membership Service Provider. A mechanism in Hyperledger Fabric that validates the identity and permissions of nodes through certificates.
CA	Certification Authorities. Entities that issue and validate digital certificates to establish secure identities.
CFT	Crash Fault Tolerance. The ability of a distributed system to continue operating even if some nodes fail unexpectedly.
IoT	Internet of Things. A network of interconnected physical devices that collect and exchange data.
PoA	Proof of Authority. A consensus mechanism based on validated identities that have the authority to validate blocks.
PoW	Proof of Work. A consensus algorithm that requires solving computational problems to validate blocks (as in Bitcoin).
BaaS	Blockchain as a Service. Cloud services that allow you to deploy blockchain solutions without managing the infrastructure.
USDGLO	Global Dollar USDGLO. Stablecoin developed on the VeChain network (oriented towards business payments).
dApp	Decentralized Application: Software that runs on a blockchain and does not rely on centralized servers.
C&C	Command and control. A server that coordinates attacks or manages malware on a compromised network.
ACK	Acknowledgment. A confirmation signal in network protocols such as TCP, indicating successful packet reception.
HTTP	Hypertext Transfer Protocol. The basic protocol for transmitting information on the web.
IIS	Internet Information Services. Web server and suite of services developed by Microsoft.
ICMP	Internet Control Message Protocol. A network protocol used to send error messages or diagnostic information (e.g., ping).
ISP	Internet Service Provider. A company that offers Internet access.
IDS	Intrusion Detection System. Technology that monitors traffic to identify potential threats.
BW	Bandwidth. The maximum amount of data that can be transferred over a network in a given time.

1 Introduction

In recent years, the growing complexity and scale of distributed systems has significantly increased the attack surface of connected infrastructures. Among the most frequent and critical threats are Denial of Service (**DoS**) attacks and their distributed variant (**DDoS**), whose frequency and sophistication continue to increase. The ability to detect these types of attacks quickly, accurately and transparently remains a fundamental challenge in the field of cybersecurity.

This project addresses this challenge by designing and implementing a **modular** and lightweight system capable of **detecting** DoS attacks in real time and immutably **recording** detected events on a public **blockchain**. The system has been designed with criteria of autonomy, scalability, and transparency, combining machine learning techniques for detection with VeChain blockchain technology for persistence.

The proposed solution is especially suitable for edge environments, such as IoT infrastructures or local networks, where centralized protection mechanisms may not be viable or unreliable.

1.1 Contributions

The main contributions developed within the framework of this project are the following:

- **Design and implementation** of a modular detection system based on the extraction of real-time traffic metrics and classification using a supervised model pre-trained on datasets labelled with different types of DoS attacks.
- Development of an autonomous **Blacklist manager** capable of classifying malicious IPs using a configurable warning system, without human intervention or the need for centralized consensus.
- Integration with the **VeChain blockchain**, which allows the recording of detected events in a permanent, verifiable, and tamper-resistant manner, thus ensuring traceability in adverse contexts.
- Implementation of a lightweight **visual interface**, embedded within the server itself, allowing the user to manage and monitor the Blacklist in real time without relying on external tools.
- Development of a traffic simulator capable of **emulating benign and malicious behaviour** under different conditions (HULK, SYN Flood, UDP Flood, etc.), which has facilitated exhaustive validation of the system.

Together, these contributions constitute a functional framework for decentralized and transparent DoS attack detection, adaptable to different deployment scenarios.

1.2 Project Structure

The project is organized into different modules with clearly differentiated responsibilities:

- **Detection module** (metrics.py, detection.py): Captures and processes network traffic in real time, calculates statistical metrics per flow, and performs instant classification using a pre-trained model. It integrates with an in-memory Blacklist manager.
- **Persistence module** (Blacklist.py): Responsible for logging, retrieving, and deleting events on the VeChain blockchain using JavaScript/TypeScript scripts executed from Python. Two versions are included: a direct implementation and a higher-level interface.
- **Server interface** (server.py, frontend.html): Provides a REST API and a web interface for querying and managing the Blacklist. It also enforces access control based on blocked IPs.
- **Logging module** (logger.py): Centralizes system logs with timestamp messages and severity levels, facilitating event analysis and debugging.

- **Traffic Simulator:** A standalone client capable of generating traffic events that replicate both legitimate users and different types of attacks. It has been key in the testing and evaluation phases.

Each component is designed to operate independently, which promotes modularity and facilitates debugging. The overall structure of the project allows for future expansion, including new attack signatures, integration with other blockchains, or deployment in containers.

2 Technological Framework

2.1 Overview of the Utilized Technology

There are several technological fields that are going to be present throughout this project. This project uses VMware workstation alongside with Ubuntu to deploy all the implementations needed.

The main technology used for the storage of the decentralized Blacklist is going to be blockchain, which offers helpful properties in relation to data integrity, transparency and security. The use of Blockchain ensures that the Blacklist remains immutable and verifiable by all related actors. The main method of validation for new entries on this list is going to be a consensus mechanism, helping leverage with efficiency and scalability.

The communication between nodes is supported by smart contracts, whose objective is to automate the process of validation, storage and access control for the Blacklist. These smart contracts help provide crucial decentralization by ensuring a secure update without the need of a specific entity with the exclusive ability to validate new entries.

Since the inception of computer networks, there have been malicious actors whose primary objective is to compromise network security and exploit systems for their own interests. Over time, these attackers have been categorized into distinct groups based on their motives, methods, and levels of expertise:

1. Black-hat hackers engage in cybercrime driven by financial gain, espionage, or the intent to cause disruption.
2. Gray-hat hackers may not harbour malicious intent but nevertheless exploit vulnerabilities without proper authorization.
3. Script kiddies depend on pre-made tools and scripts, typically lacking deep technical knowledge or understanding of the attacks they carry out.
4. Hacktivists employ cyberattacks as a means of protest or political activism, aiming to generate social or political impact.
5. Among the most dangerous are state-sponsored attackers and cyberterrorists, who target critical infrastructure and national security with considerable resources and strategic objectives.

As cyber threats continue to evolve, so too do the techniques used to detect and prevent them, making cybersecurity an ongoing race between attackers and defenders.

Of the many modern malicious tactics DoS stands out as one of the most widespread, as well as one of the most damaging. Cybercriminals seek financial gain, hacktivists protest organizations or state-sponsored organizations aim to disrupt critical infrastructure, all with a similar principle denying the objectives system functioning ability by overwhelming their infrastructure and devices. Doing so by massively increasing the targets traffic can lead to the crash of websites, online services or simply cause (in occasions immeasurable) economic and financial turmoil. Attackers constantly develop ways to elude active security measures such as the use of botnets [1], amplification techniques and even AI-powered strategies that maximise impact. This makes developing effective detection and mitigation methods almost unfeasible for cybersecurity professionals, as they must constantly adapt to new threats in the ever-changing digital landscape.

Over the years DoS attacks have become complex, sophisticated and harder to stop. They adapt to the changing technologies and exploit specific weaknesses of their protocols, either at the network or the application level. 5G accentuates these issues as faster rates and better connectivity provide with opportunities to attackers.

This project intends to explore both these worlds and offer ideas to improve attack prevention by decentralizing a Blacklist built on the idea of Machine Learning capabilities to detect through traffic filtering, rate limiting, and behaviour-based detection. The aim is to understand how to better protect systems in an increasingly digital and interconnected world.

2.2 Introduction to Blockchain

The concept of blockchain goes back to the early 90s with Harber and Stornetta. They proposed a system for securing digital documents via linked timestamping, effectively creating the first “chain-of-blocks” [2]. Later that century, Bayer et al. proposed incorporating Markle trees [3] as to improve the efficiency of the system; this concept is regarded as the base for the famous cryptocurrency “Bitcoin” [4].

At the end of the decade there was already a discussion on the practical prospects of this technology. Precedents like B-Money [5] offered the first idea of cryptocurrency, which implied the use of virtual currency based on the ideas of decentralization and cryptography. This technology uses blockchain to record all transactions. The structure of each block is, as drawn in figure 1, composed of [6], [7]:

1. Hash block: Made up by the hash of previous block’s header, and its own header’s hash.
2. Merkle Root: Hash of the Merkle tree containing all transaction hashes. This element, although common, is optional in most blockchains with important exceptions such as Bitcoin.
3. Timestamp: Indicating the exact moment the block was created.
4. Nonce: Value used for consensus to satisfy validation conditions. There are several mechanisms such as PoW, PoS, PoA, etc.
5. Difficulty Target: Used to define the difficulty level of the mining process.
6. Body: Includes transaction data and transaction count. It can include more metadata depending on the technology, like smart contract code execution (platforms like Ethereum), State information, Sidechains and Sharding (improves scalability, e.g. Polkadot or Ethereum 2.0) or Permission type (differentiates between permissionless blockchains, e.g. Bitcoin, Ethereum, and permissioned blockchains, e.g. Hyperledger Fabric).

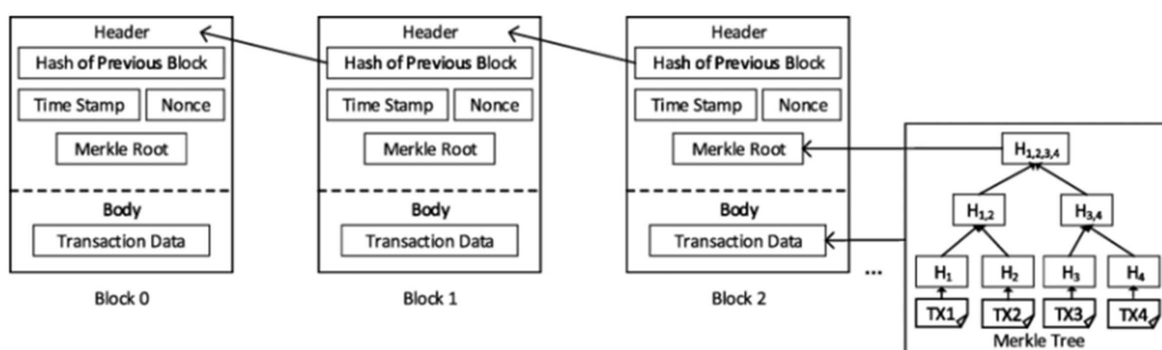

Figure 1. Structure of a Block in a Blockchain

Blockchain is based on these blocks, but the technology is not exclusively defined by them. *Node* is the name given to the actors that participate on the network. These nodes are devices (hardware) that connect to the network and maintain it. They oversee the processes of validation and rely on transactions and blocks as shown in figure 2, HLF workflow.

Transactions are also an important factor in blockchain. A transaction is defined as the process of transferring information between parties (nodes) in the blockchain network. A transaction is a fundamental unit that can be recorded on a block, forming the basis for a blockchain ledger.

More advanced blockchains inherit characteristics of all these technological improvements. The birth of blockchain is usually linked with Satoshi Nakamoto and the origins of Bitcoin. Their paper *Bitcoin: A Peer-to-Peer Electronic Cash System* presents the idea of using several mechanisms and new concepts to create a decentralized, secure, unsupported and unique kind of currency. Bitcoin makes use of a PoW consensus method, based on the idea of using the hash function to ensure integrity [8]. It also introduced a decentralized ledger as a temper-proof chain. The first block was "mined" in January 2009.

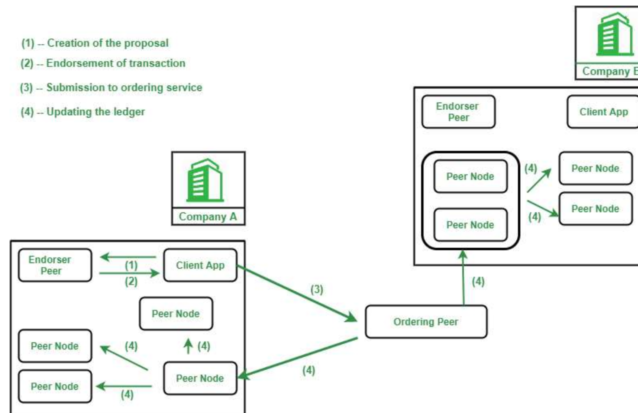

Figure 2. Workflow of Hyperledger Fabric [6]

Modern blockchain networks evolve these ideas and introduce new ones. Smart contracts originated in the 1990s [9] but were implemented first in blockchain with the introduction of the new cryptocurrency Ethereum [10]. Ethereum Turing-complete Virtual Machine (VM) expanded Blockchain's use case beyond currency.

Next advancement was the exploration of private blockchains and the definition of permissioned Blockchains. HyperLedger Fabric is the first one of the platforms that characterize these advancements. It offers a blockchain network meant to improve scalability as well as improve privacy and confidentiality in enterprise use cases. It achieves this through its modular architecture, allowing organizations to tailor their blockchain solutions with features like permissioned access, pluggable consensus protocols, and smart contract support. Hyperledger Fabric's focus on enterprise-grade scalability is bolstered by its ability to handle hundreds of transactions per second, making it suitable for diverse applications ranging from supply chain management to financial services. This platform represents a significant step forward in blockchain technology, particularly in addressing the complex requirements of large organizations seeking secure and efficient decentralized solutions [11].

2.2.1 Blockchain Nodes

As defined in the previous point, nodes are Hardware components that build, maintain and interact with the blockchain. These components tend to be computers but are not limited to them. Any electronic device that interacts with the blockchain is a node.

Nodes oversee various aspects of the network. They maintain copies of the ledger, keeping in storage information of the entire blockchain. This helps providing immutability of the blockchain, adding security aspects key to this project as well as making the process easily scalable. Nodes are also in charge of the validation process, making proposals and acknowledging transactions. This process is based on a majority consensus, helping in the authentication of each block and eliminating the only validator. Having a consensus validation allows decentralization, which is crucial for ensuring that no single entity has control over the entire network, promoting transparency and enhancing security within the network.

There are several types of nodes [12] [13]:

1. **Full Nodes:** Maintain the entire blockchain ledger and validate transactions and blocks.

2. **Light Nodes:** Store only a portion of the blockchain, relying on full nodes for validation.
3. **Miner nodes:** Solve complex cryptographic puzzles to create new blocks.
4. **Pruned Full Nodes:** Store only the most recent blocks, discarding older ones after validation.
5. **Archival Full Nodes:** Keep a record of all transactions from the genesis block.
6. **Authority Nodes:** Used in permissioned blockchains to validate blocks.
7. **Master Nodes:** Perform specialized tasks such as governance or additional validation functions.
8. **Staking Nodes:** Validate and secure the network by staking cryptocurrencies.
9. **Lightning Nodes:** Facilitate faster, off-chain transactions in networks like Bitcoin's Lightning Network.
10. **Super Nodes:** Provide enhanced processing power and connectivity, often handling high-volume transactions or acting as network relays.

2.2.2 Blocks in a Blockchain

As mentioned, blocks are the foundational components of a blockchain. They are used to store data, such as transactions, or in the case of this project, metadata regarding network users' information. Each block is linked to the previous adjacent block using hash functions and other cryptographical measures, forming a chain. Blocks are designed to ensure transparency, traceability and immutability, making the blockchain a greatly trusted technology for decentralized data logs.

The storage is done in a structured manner, such that each block is divided in specific components that compose the entire structure of the block. Depending on the specific technology utilized there can be different components in each block. For this paper, only Bitcoin due to its historical importance, Ethereum for similar reasons, Solana and Hyperledger Fabric are going to be analysed.

Common structures of blocks in all these technologies are:

1. **Block Header:** Variable in the different technologies but usually containing metadata regarding the block. Usually they include fields such as:
    a. Previous/parent Block Hash: Used to reference the hash of the previous block, linking together the blockchain.
    b. Timestamp: Immutably determines the time of the block's creation.
    c. Block Identifier: Identifies univocally the block within a blockchain. Usually determined by parameters within the block, like its own hash.
2. **Block Data:** Stores information of operations, transactions, etc. Typical components are:
    a. Transactions: List of different operations or transactions recorded in a specific block.
    b. Smart Contract Data: In the case of blockchains that use this technology like Ethereum or Solana, inclusions of the utilized code for specific smart contracts on addition to the execution results is typically included.
3. **Consensus Mechanism:** Used by the blockchain to enforce an agreement on the ledger state, meaning the inclusion of new entries. They are utilized in substitution of human auditors.
4. **Network Metadata:** Additional information regarding the blockchain network, its participants and the state of the blockchain. It is not included in the actual transactions of the blockchain, but provides context, structure and possible operational details to maintain the network. It can be stored on-chain (directly in the blockchain) or off-chain (in external systems).

Once a block is added to the blockchain it becomes inalterable. This is achieved using hash functions as previously mentioned. Examples of this are SHA-256 for bitcoin or Keccak-256 for Ethereum.

2.2.3 Types of Blockchain

Blockchain technology is usually divided in three main types [14]:

The first and most common type to be discussed in this framework is *public blockchains*. They are characterized by being open and decentralized; public blockchains are accessible by anyone at any time, making them accessible to everyone. The access in this blockchain is universal, meaning whoever is interested can participate with the validation of transactions.

No authority monitors the network, ensuring the trust of the different active parties without an intermediary. There is a high level of security, as there is a strong focus on security and the study of adequate consensus mechanisms, making it extremely hard for malicious entities to approach the system. Anonymity is present throughout the whole procedure, making the identity of all nodes invisible.

They also have some disadvantages such as low scalability, meaning greater difficulties when integrating new nodes, a high energy consumption and a low latency per transaction due to the elevated number of affected nodes in the process of validation.

Some examples of this technology are Ethereum, Bitcoin or Solana. Typically, this technology has been given the use of cryptocurrency exchanger, although there is clear potential on the implementation of new ledgers for different uses such as cybersecurity or banking registries.

The second type is labelled as *private blockchains*. Private blockchains are controlled by an organization, limiting its access. This model is typically found in enterprise environments where confidentiality, access control and efficiency are priorities. Some examples of this network type are HLF or R3 Corda [15], used in healthcare, finance or logistic sectors.

The third and final type is *consortium blockchains*. In this model, transaction management and validation do not fall under a singular entity but in a predetermined group of organizations. This group shares common interests offering equilibrium between decentralization and efficiency. It is ideal for interenterprise applications like private-public cooperation or supply chain management.

2.2.4 Consensus Mechanisms

Consensus Mechanisms are algorithms used by different entities within the blockchain to concord the main state of ledger´s state in the network [16]. This procedure must be done in a decentralized and secure manner. Depending on the network type there are a wide variety of consensus mechanisms:

- **PoW** – Proof of Work:

Typically used in cryptocurrency-oriented networks. It is supported by “miners” who resolve complex cryptographic algorithms to add new blocks. PoW provides high levels of security but with large energy spendings.

- **PoS** - Proof of Stake:

Used in Ethereum 2.0, Cardano… Based on the idea of placing cryptocurrency as liable for the appropriate acceptance of new blocks [17]. It produces a more energy efficient model but can be inclined to a more centralized network.

- **DPoS** - Delegated Proof of Stake:

Used in EOS, Tron, etc. Here, users choose a representative to validate blocks. It improves scalability but, as with PoS, it reduces decentralization.

- **PoA** - Proof of Authority:

Used in HyperLedger Besu [18] and other private networks. Based on trusted authorities preselected by the network.

- **RAFT** - Dependable, Replicated, Redundant, And Fault-Tolerant & PBFT-Practical Byzantine Fault Tolerance:

RAFT is based on a similar concept as PoA but with the use of a leader. Typical of private networks as HyperLedger Fabric.

2.2.5 Transactions and Transaction Process

The basic unit for information exchange in a blockchain network is the transaction. A transaction represents the exchange of data between participants of the network in a decentralized and secure manner.

Several actors are considered for a transaction to be made. Some of the most typical ones are:

- **Sender & receiver:** Involved in the transaction with their public keys.
- **Data:** Information to be transmitted.
- **Digital Signature:** Sender cryptographic validation.
- **Hash:** Identifies the previous transaction.

When creating a transaction, the emitter needs to provide a signature with the use of their private key, ahead of distributing it through the network nodes. Once it reaches the nodes (in the case of a permissioned blockchain the authority node) the signature is validated and a checkout for funds or permits is done.

If valid, the transaction is incorporated inside a block. Once a block is created and confirmed through the specific consensus mechanism process it is added to the blockchain. Transactions are immutable and accessible to all participants of the network.

2.2.6 Smart Contracts

A smart contract is the program in charge of executing automated instructions when specific, predefined trigger criteria is due. The of a smart contract logic is defined within the paradigm of if-then-else, granting deterministic executions.

These codes can be developed in any programming language. As to exemplify we can look at Ethereum or VeChain as they typically use Solidity or HyperLedger Fabric (HLF), which accepts any kind of language but is designed to use mainly Go, Node.js or JavaScript.

First steps are defining and writing a smart contract in any programming language compatible with the specific blockchain. Once done, the source code must be compiled through a process to convert the contract into an executable format called a **bytecode**. The bytecode is used by the blockchain as it contains the instructions to be interpreted in the corresponding nodes (such as the EVM in Ethereum or the Chaincode Runtime in Hyperledger Fabric). This ensures uniform execution for all participants in the blockchain.

Different technologies use different methods to implement this **Chaincode**. HLF uses a peer Chaincode, Ethereum based blockchains use a wide variety of methods to implement this like the web3.eth.Contract.deploy() [19][20].

Users interact with the contract in transactions, updating the distributed ledger. Different networks make use of different technical specifications to integrate the user and the smart contract. Oracles are services used by smart contracts to access external data not stored in the blockchain [21].

On a blockchain, once a smart contract is deployed, its code cannot be changed. This ensures trust and security. If an update is needed, a new contract must be deployed at a new address, and systems must be updated to use the new version [22].

2.3 HyperLedger Fabric

HyperLedger Fabric (HLF) is one of the most popular HyperLedger platforms. If has been designed by the Linux Foundation to pursue the objective of creating business-directed blockchain technology. HLF stands out as an open source, modular and expandable system. With a main goal of deployment and operation of permissions blockchains, HLF operates making use of standard programming languages such as **Go** to execute distributed applications, without the need of a native cryptocurrency. The design supports consensus modules, making the network compatible with a variety of plausible uses or confidence models. HLF also introduces a different blockchain design meant to improve safety as well as performance. This design is supported on **private channels** to create communication paths between different entities within the network. When tested, it has been capable of executing over 3500 transactions per second, with a latency below a second and the scalability up to 100 nodes [20].

2.3.1 Network

In HLF, as in most blockchains, the network is the name given to the global infrastructure that interconnects all the components of the blockchain. This infrastructure is defined by its architecture and understanding the idea of creating a modular blockchain permits the customization of most aspects of the network [20].

HLF is easily scalable, supporting increasingly big networks, thousands of nodes and a huge number of transactions per second [23]. It also offers privacy and confidentiality within the network with the implementation of channels, permitting the creation of a private network within the public one [20]. Moreover, it offers adaptability to its users, allowing the modification of the network to adjust to various possible use cases, from consensus to data storage [24].

The non-use of cryptocurrency offers an ideal environment for developers and testers to experiment on a practical blockchain without the need of without the need of dealing with financial regulations, transaction fees, or economic incentives.

2.3.2 Configtxgen

In Hyperledger Fabric, blockchain network configuration is a critical process that defines the organizational structure, channels, and governance rules. This section describes the use of the configtxgen tool, the format of the **configtx.yaml** file [25], the use of Protobuf for configuration serialization, and the policies applicable to channels.

The configtxgen tool is essential for initializing and configuring a Hyperledger Fabric network. Its main function is to generate configuration artifacts such as:

- **genesis.block:** genesis block of the system.
- **channel.tx:** channel creation transaction.
- **anchor.tx:** transactions to define the anchor peers for each organization.

The configtx.yaml file is the main input for **configtxgen**. It defines several key sections, including:

- Organizations: defines the participating organizations, their certificates, and local policies.
- Orderer: defines the ordering system parameters, including consensus type (e.g., RAFT), policies, and authorized organizations.

- Application: describes how application channels are configured, including specific policies.
- Profiles: they are used to generate the aforementioned artifacts, grouping configurations under a name that can be invoked with configtxgen.

Each section incorporates access control policies that govern which entities can perform administrative operations, such as updating the channel or changing configurations.

2.3.3 Protobuf

Network configuration, as well as block and transaction information in Hyperledger Fabric, is serialized using Protocol Buffers (Protobuf). This compact and efficient serialization facilitates interoperability between components written in different programming languages.

Files generated by Configtxgen, such as genesis.block or channel.tx, are encoded in Protobuf. Therefore, for inspection and modification, it is necessary to use tools such as Configtxlator, which allows conversion between Protobuf and JSON [26].

2.3.4 Channel Policies

Policies in Fabric are expressions that define which identities are authorized to perform certain operations within a channel [27]. Policies can be applied at multiple levels:

- **Readers**: They can read blocks from the channel.
- **Writers**: They can submit transactions to the channel.
- **Admins**: They can make configuration changes.
- **Endorsement**: They must approve a transaction before it is validated.

Each policy can be defined using Signature, ImplicitMeta, or OutOf syntax:

- **Signature**: Explicitly defines the required identities.
- **ImplicitMeta**: Used at the channel or system level to infer policies from subcomponents (e.g., most organization administrators).
- **OutOf**: Boolean logic that allows for complex combinations of policies.

2.3.5 Nodes

Nodes are the key components of the HLF structure. They permit the execution, validation, transaction consensus and the storage of the information. There exist several types of nodes, each with specific roles within the network:

To begin with, *Organizations* are entities that control a diverse group of nodes, with its own identities and Access policies. Each Organization counts with their own MSP (Membership Service Provider), which is the component responsible for managing the digital identities of that organization's participants. The MSP defines who is considered a legitimate member of the network and allows for the establishment of authentication, authorization and access control rules. Additionally, each organization can define its own consensus and validation policies, enabling decentralized yet structured management within networks like Hyperledger Fabric.

Secondly, Peer Nodes are the main participants of the network. They execute and validate transactions, as well as store a copy of the Ledger. There are 2 types of peers:

- **Endorsing peers,** in charge of executing transaction proposals as well as signing the valid ones.
- **Committer Peers,** whose main purpose is to validate transactions ordered by the orderer nodes and storing the transaction block as well as updating the ledger.

Thirdly, **Orderers** are responsible of ordering proposed transactions and ensuring their propagation throughout the whole network.

Channels are private sub-networks within a HLF network where any number of participants can exchange data in a private and secure manner.

Another component of the network is the **Chaincode**. Chaincodes are the files that define logistics behind transaction. They can be written in any programming language, although typical ones are Go or JavaScript.

Finally, the **ledger** is a register that accounts for all successful transactions. It is composed of two main parts: a) *World State* stores data's current state and b) blockchain, which is in charge of maintaining a complete register of all transactions.

2.3.6 MSP & Identity Management

In a HLF network, each participant possesses an unequivocal digital identity identifier. Typically, it is encapsulated in a **X.509 digital certificate** [28] or **LDAP**. This identity identifier is used to determine the permissions possessed by the node and/or label each different component in specific groups, such as peers belonging to an organization. As a verifiable authority of this IDs there is introduced the concept of **Membership Service Provider** (MSP) [28].

MSP as shown in figure 3 defines each organization in the network. It is just a directory containing a group of files to identify different actors within the network. It plays a crucial role in defining identities and roles for different participants [29]. They manage certificates and cryptographic materials that authenticate and authorize entities like Root CAs and Intermediate CAs. Each organization has its own MSP, constructed to define the authentication policies in use, authorise actors (peers, orderers…) and the trusted certificates [29].

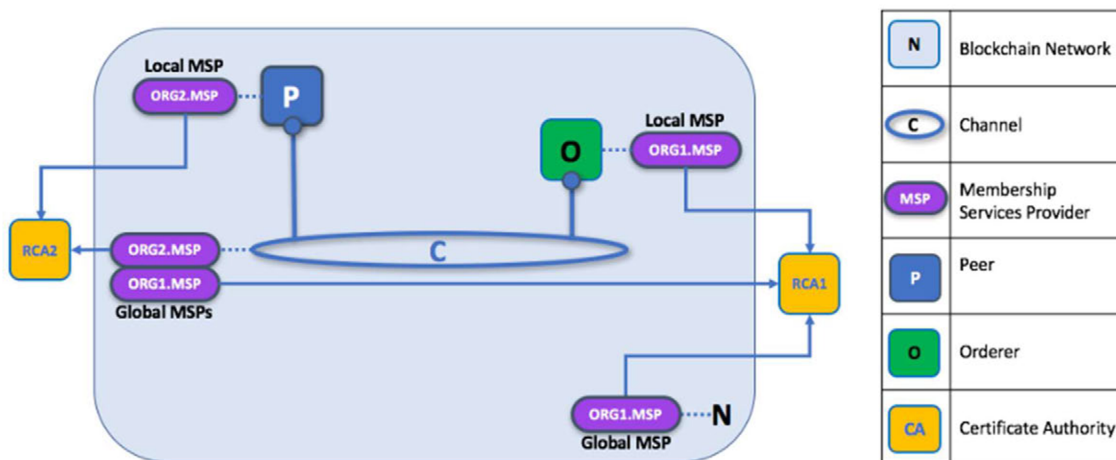

Figure 3.Membership Service Provider (MSP) - HLF [30]

2.3.6.1 *Attribute Based Access Control (ABAC):*

Hyperledger Fabric makes use of the **ABAC model** to control the access to the resources [31]. It is based on attributes associated to the user's identity [23]. These attributes are included in each certificate associated to the different actors of the blockchain. The attributes define access politics in a detailed and flexible manner.

A use case to illustrate this is a policy that allows only users with a particular attribute to invoke certain functionality of the Chaincode. For instance, a defined user is the only one capable of modifying specific entries. To do this an attribute is created with unique values and modifying features that be active or inactive. An organization has a line defined as the one in the figure 4. This attribute is incorporated to the **organizations certificate** and validated by the **Chaincode**. The Chaincode is in charge of validating and identifying the fields value before allowing the execution of operations.

When an organization stablishes a policy as intended where only certain actors can make modifications, it is imperative for concrete entities to have the attributes corresponding configured in its certificate. Through these attributes and processes the network ensures only authorised entities and participants can access critical system functions.

```
AND('Org1.member', 'attr:modificator == "active"')
```

Figure 4. Attribute modification for Org 1

As shown in figure 5, the fabric-ca-client register command allows you to register a new identity with the Fabric CA server. In this case, an entity named editorEntity is registered with the password editor, classified as client. Additionally, it is assigned the custom attribute modifier=active with the suffix: ecert, indicating that this attribute should be included in the entity's certificate. This attribute will later be used by Attribute-Based Access Control (ABAC) policies to allow, for example, the modification of specific Chaincode entries [31][32].

```
fabric-ca-client register \
  --id.name editorEntity \
  --id.secret editorpw \
  --id.type client \
  --id.attrs 'modificator=active:ecert'
```

Figure 5. Entity register with personalized attributes

2.3.6.2 *Role Based Access Control (RBAC):*

Hyperledger Fabric also allows the implementation of the **RBAC** (Role-Based Access Control) model to manage access to resources [33]. This model is based on predefined roles assigned to users within the network. These roles are represented as **attributes in the digital certificates** of the different actors of the blockchain. Access policies can then be defined according to the assigned roles, enabling a more structured and hierarchical **permission management** [33] [34].

This perspective is useful for enterprise entities, where there is a clear differentiation of functions such as administrator, auditor, editor… To illustrate this, we find policies that can only permit users with editor role to make active changes in the network or allow readers to only have read permits, without any possibility of content modification. This role management can be made through MSP and applied during transaction validation through Chaincode [35].

In combination with the **ABAC** model, **RBAC** permits the construction of more robust access control systems, making roles behave as filters and attributes as specific rules. HLF eases this integration by allowing multiple attributes and roles in the Fabric CA emitted X.509 certificates.

2.3.7 Channels

HLF makes use of different mechanisms to ensure the privacy in transactions. Probably the most notable of this mechanism is called **channel**. A channel is like a VPLS, allowing direct communications between private devices through a common network. A channel is defined as a **subnet** within the main blockchain network [35]. This subnet is only accessible to authorised participants and counts with an associated private ledger. This is designed to ensure sensible information remains private and confidential whilst being transmitted by the involved participants.

2.3.8 Transactions

Every proposed **transaction** needs to be validated by the **endorsing peers**. Once approved by a determined number of nodes it is formalized and registered by the orderer nodes. In HLF, transactions are used to update the ledger status, as well as executing Chaincodes. This is defined in the transactional process, designed to ensure security and consistency whilst maintaining privacy.

In each transactional process, different entities take part on one or various actions. Each of those actors are defined as follows:

- **Client:** Actor of the application layer, this entity is typically the solicitor of the transactions.
- **Endorsing Peers:** Nodes that validate transactions. This validation is done by executing the Chaincode for the proposed transaction and generating an approval "endorsement".
- **Orderers:** They order transactions and assure the diffusion inside the network. They make use of the consensus algorithms to decide this order.
- **Committing Peers:** Once ordered, transactions are sent through the network. Committing peers validate this transaction and once enough have done it they register it in the ledger.
- **Chaincode:** Defines the business logic of the transactions.

The simple steps of a transaction are:

1. **Application:** A client solicits a transaction to one of the endorsing peers, specifying which operation of the Chaincode to invoke.
2. **Execution & Endorsement:** Endorsing peer executes the corresponding invoke, verifying its validity within the norms stated in the Chaincode. If valid, it generates an endorsement and returns it to the client.
3. **Distribution:** The transaction is then sent to the orderers for its integration in blocks.
4. **Ordering:** Orderers order the transactions with the appropriate consensus mechanism (RAFT, KAFKA…) and distribute the resulting blocks through the network to the different peers of the specific channel.
5. **Validation:** The different committing peers validate (or not) the transactions to ensure no conflicts. They then update the ledger with the valid ones, granting immutability.

There are two main transaction types [36]:

1. **Deploy transactions:** They are used to create a Chaincode inside a channel.
2. **Invoke Transactions:** Once a Chaincode is deployed, access to its methods is done by invoke transactions.

HLF is capable of processing from over **3500 transactions** per second to even 20000 if enhanced properly through optimization processes [37]. The system presents latencies up to milliseconds for common configurations and facilitate the inclusion of more peers, permitting scalability and manage great transaction volumes maintaining performance [38] [39]. The system permits up to over **100 peers** [37]. Tests like the ones done by show functionality with networks of up to 32 organizations and 128 peers, including a channel interconnecting each for a total of **325** simultaneous **working channels** with a latency of still under a second [40].

2.3.9 Chaincode

Like a smart contract, **Chaincode** implements the **logic** behind every transaction of a HLF blockchain network. It is executed **isolated**, typically by **dockers** inside the endorsing peers but outside the validation process [41].

It can be defined as code built to specify handle business logic within the network. Chaincodes can be interconnected, being able to invoke others. It can be in charge of all the different functionalities of the network such as storing information, consulting it or modifying/erasing ledger information.

Some of the most notable characteristics of a HLF Chaincode are [42]:

- **Programming language:** Adaptable, it can be programmed in any language, but HLF developers typically use Go, Node.js or JavaScript. These 3 languages are the ones HLF support team works with and the ones most GitHub repositories are written in.
- **Docker:** Chaincode is executed in isolated environments to provide security and stability to the system.
- **Ledger integration:** Chaincodes final purpose is to edit the ledger, adding, deleting or modifying entrances. This is done with the help of API´s such as Fabric Chaincode Shim API.

Some key steps on typical Chaincodes are:

- **Development:** Programmers code the files that describe the Chaincode in the chosen programming language.
- **Installation:** Once the file/s are ready, administrators install the Chaincode in the different endorsing peers of the channels that are going to make use of this ruleset.
- **Approval:** Each organization that takes part of the channel's communication needs to review the Chaincode. Once it is approved, by default a simple majority, it is approved. After approval the endorsement politics are stablished.
- **Commit:** After approval, the channel´s definition is committed in the channel.
- **Invoke:** Clients execute functions of the Chaincode by invoking them.

2.3.10 HLF Consensus Mechanism

Several consensus mechanisms are stablished, allowing an adequate protocol selection for a variety of business models and confidentiality requirements. HLF does not use traditional consensus methods sus as PoW or PoS; it instead focuses on **order** of transactions, meaning the process of organising and **sequencing** the transactions on advance, followed by the **ledger confirmation** [43]. This process is done by **orderers**, which use two main possible protocols to process valid transactions. **Raft** is an up-to-date protocol based on the designation of a leader node and replicas, meant to be more flexible and to have a simpler maintenance. It is recommended for versions of fabric 1.4 and up [43]. **Kafka** was the previous protocol, not recommended in more modern networks. It is more complex and offers a harder maintenance. Kafka is based on an inter orderer message system helped with brokers to clean and sustain the communication.

Apache Kafka is a distributed message system utilized to order transactions. It is the original orderer algorithm, typical of older versions of HLF. It proposes a failure tolerance system based on clusters of orderers that instantiate a leader-follower paradigm. Its complexity has led to RAFT overtaking it as the principal orderer algorithm. It is a robust implementation with a high latency and not so good performance [43].

On a positive note, Kafka stands out for its capacity to manage **great volumes** of messages in a distributed manner. This message management makes it ideal for systems that require high trust delivery levels [44]. However, this strength also comes at a cost in terms of performance, as synchronization between cluster nodes and consensus management generate considerable overhead [43] [44].

This is the main reason as to why Kafka, which pioneered as a solution for transaction ordering, is currently being abandoned. Its complexity and latency have made HLF and most other networks pivot towards faster, more efficient algorithms, such as RAFT.

RAFT is the new adopted algorithm for transaction ordering. It is a **CFT tolerance-based algorithm**. It also operates with a leader and follower model. It is far simpler than Kafka and as such it offers better throughput, lower latency and less operational complexity. It works under a leadership model where one of the nodes acts as a leader and the rest as followers.

The leader is responsible for receiving all transactions, ordering them and replicate them as to distribute transactions consistently between followers. This simplifies consensus logic, preventing complexities associated to KAFKA.

The **simple** design and ease of implementation translate in **cost reduction** and simpler maintenance in distributed environments. On top of that, RAFT offers a superior throughput, reducing latency. RAFT is ideal in environments where simplicity, availability and speed are vital.

2.3.11 Ledger & Data Storage

The ledger is the core component of the network. It stores all data related to the blockchain. It is composed of two main components [45]:

1. **Blockchain:** Immutable log of all transactions, stored as a file-based system.
2. **The world State:** A key-value database representing the states of all assets. It is updatable [46].

2.3.12 Docker & Docker-Compose

HLF makes high use of containerization mainly with **Docker**. Its main purpose is to **manage** and **isolate** different components. This workstyle provides high **flexibility** when deploying networks as it can easily be replicated and scaled across diverse environments. Each component runs a differentiated container, ensuring separation and simple maintenance.

Docker Compose also takes an important role in HLF development. It is used to order the different containers that construct the network. Docker-compose.yaml is a file meant to define multiple services as peers or orderers with a simple instruction. This procedure eases the fast creation of environments, replicable and portable.

2.4 VeChain

VeChain is a **public blockchain** platform focused on **enterprise** solutions to improve supply chain management, product traceability and data verification [47]. Founded in 2015, it has steadily consolidated as one of the industries leaders in 2.0 industry, intending to combine blockchain technology and IoT [47].

The main network is called **VeChainThor**. VeChainThor is optimized for real world applications, enabling business to develop decentralized solutions without the need of building a private network from scratch [47].

Unlike more generalist blockchains, VeChainThor incorporates **native features** such as identity management (VIDs), metadata for traceability, and multi-task transactions (Multi-Task Transactions – MTT), which allow multiple actions to be executed in a single operation [49] [48].

For information regarding transactions, smart contracts, accounts, blocks or tokens there is a **deployed online web service** meant to ensure transparency of the blockchain [50]. It allows access to data of the mainnet as well as the testnet. On a similar manner, VeChain Stats take all this information and displays it including market information such as market caps, price or authority nodes for VET, VTHO and USDGLO [51].

2.4.1 Consensus Algorithm

VeChain utilizes **PoA 2.0**, ensuring safety and speed in block construction. PoA 2.0 is a version of its predecessor that implements a **Byzantine Fault Tolerance** and **Nakamoto Consensus**. It implements additional security mechanisms such as the **random validator** selection and the **committee validation** [52] [53]. All of this is done while perfecting the failure acceptance, tolerating up to a third of the total nodes being malicious without failure. Scientists and expert investigators from the VeChain team designed it for enterprise applications [54].

The introduction of Proof of Authority (PoA) 2.0 represents a significant advancement in consensus algorithm design, as it blends the deterministic finality of BFT protocols with the decentralization principles of Nakamoto-style consensus. This **hybrid model** mitigates common issues found in Proof-of-Work and traditional PoA mechanisms, such as energy inefficiency and centralized trust. As outlined by VeChain technical whitepaper and corroborated by blockchain security research, PoA 2.0 reduces the time to finality while ensuring resilience against various attack vectors, including long-range attacks and double-spending scenarios [55].

Furthermore, the **random selection** of block proposers and the **validation committee** increases unpredictability, **reducing** the likelihood **of targeted attacks** on specific validators. These innovations make VeChain particularly suitable for sectors requiring **high reliability** and **trust**, such as supply chain management, logistics, and pharmaceutical traceability. Academic literature emphasizes the importance of secure consensus mechanisms tailored for enterprise-grade blockchain applications [56], and PoA 2.0 stands out by providing a balance between throughput, scalability, and fault tolerance.

2.4.2 Token System

VeChain uses a dual-token system to differentiate the market value from the operational costs:

- **VET (VeChain Token):** Serves as a reserve of value for the blockchain in market as well as an automated means of generating VTHO tokens, the base VTHO generation rate is 0,000432 VTHO per VET per day [57]. The total supply of VET is capped out at approximately 86.7 billion tokens, granting scarcity and supporting long-term value [57].
- **VTHO (VeThor Token):** Used as gas payment. Auto generated by VET.

This structure allows enterprises to use the network without the exposure to the VET market volatility.

2.4.3 Tools & SDK

VeChain provides a set of robust development toolkits designed to facilitate integration with real-world applications. Mainly focused on developers and enterprise users, these tools aim to simplify the deployment and management.

VeChain ToolChain™ is a **BaaS** platform designed for enterprise blockchain integration. It is conceived to incorporate blockchain in business processes without the need of a profound technical expertise.

It is widely used in the food, automotive or fashion industries. Within the toolchain we find several identarian characteristics [57]:

- Ready to use **graphic interfaces** with prepared API`s.
- Product **traceability** templates.
- **Cloud infrastructure & IoT support**.
- Complete data **privacy** and permits **control**.

VeChain testnet is a blockchain that intends to be a **testing environment** for developers to experiment with. Its main functionalities are identical to the ones offered by the mainnet [58]. Some of its more important characteristics are:

- **Test tokens** mirroring VET and VTHO with identical conditions of supply and generation but with no intrinsic value within them.
- **Faucets** are wallets made to hold testnet tokens. Developers use them to carry transactions.

VeChainThor Wallet stands as the official mobile VeChain wallet. Developed for various mobile O.S. (android & IOS) it offers [59]:

- Secure token storage.
- Multiple account management.
- Direct access to dApps based on VeChain.
- Transaction explorer
- Smart contract creation
- Compatibility with wallets such as ledger.

Connex is a **JavaScript** library used as a socket to communicate between **dApps** and **VeChainThor** blockchain. It is licensed by the GNU Lesser General Public License v3.0 (LGPL-3.0). Some of the more important characteristics are [32]:

- Access to accounts and signatures.
- Transaction management
- Smart contract management.

Connex is an essential component in the construction of user interfaces and smart contracts.

Sync (current version Sync2) is the official desktop VeChain wallet. It is used mainly to manage tokens and associate the different wallets with the dApps. Characteristics are as follow [60]:

- Similar to MetaMask, adapted to VeChain.
- Support Connex and VET/VTHO (and many more blockchains).
- Secure transaction signing.
- Integrates with VeChainThor Wallet by QR code usage.

It is encouraged for trial environments and dApps development.

Thor Devkit is a library for developers to directly interact with the VeChain blockchain directly for the applications. It is available in various languages such as TypeScript, Java, C# or Python. This kit allows [61]:

- Building and signing transactions.
- Managing various accounts, performing operations like private key generation or mnemonic words handling.
- Hash and public Key management.

2.4.4 Use Cases

VeChain can be found in diverse industries and enterprises. Some of the most notable ones are [62]:

- **Supply Chain Management:** VeChain allows products to be tracked from their origin to the final consumer, ensuring authenticity, quality, and traceability. This is especially useful in industries such as food, pharmaceuticals, and luxury goods, where transparency is key to preventing fraud or counterfeiting.
    - Walmart China: Uses VeChain to trace food products from their origin, allowing consumers to verify product information using QR codes.
    - DNV GL – MyStory: Platform that uses VeChain for brands to share verifiable data on the sustainability and traceability of their products, including the origin of materials and ethical processes.
- **Authenticity and Counterfeit Prevention:** Thanks to its blockchain technology, VeChain helps certify the authenticity of products such as wines, high-end clothing, automobiles, and electronic components, preventing the circulation of counterfeit products in the market.

  - LVMH: Implements VeChain to authenticate luxury products such as Louis Vuitton and TAG Heuer, preventing counterfeiting.
  - H&M (COS and Arket): They use VeChain to track the traceability of sustainable wool in their garments, certifying the origin and manufacturing process.
- **Automotive and Connected Vehicles:** VeChain can store relevant vehicle history data, such as maintenance, repairs, or mileage, providing an immutable record that increases confidence in sales and after-sales transactions or after-sales service.
  - BMW – VerifyCar: Platform developed with VeChain to verify the maintenance history of used vehicles.
  - Renault – Digital Maintenance Book: Immutable record of vehicle repairs and maintenance developed with VeChain and Microsoft.

2.5 Denial of Service

2.5.1 Dos

Denial of service is a **cybersecurity attack** based on **saturating** different **network** components or **systems** to prevent them from properly executing their designed tasks. This is achieved by massively sending malicious traffic to saturate the resources of the victim.

DoS attacks are characterized by having a **unique terminal** or **IP address** to generate the petitions. They can send up to millions of requests, flooding the service. This can last from minutes to days, creating prolonged issues. Most of the times the incentive is economic or political, where the attackers seek to benefit from the victim or its adversaries. Occasionally, a huge attack is directed at random services without a specific motive on the target. DoS can be done to enhance Hackers or Organizations reputation and visibility. Consequently, there can be major losses for both businesses and people. **Disruption** of web services leading to collapse of the utilized technology can bring about serious economic losses and reputational damage [63].

2.5.2 DDoS

Newer versions are the **distributed denial of service attacks**. Built on the **same principles**, DDoS maintains its objective of flooding services as to incapacitate them. In this case the attack is coordinated from a cluster of infected or associated computers, provoking a complex cluster form of infectious computers (**botnet**). This distributed origin makes mitigation more complex as it is harder to find the origin of the attack (**Botmaster or Header**).

The malicious attack is typically structured in such manner as for the botnets to attack simultaneously. The typical attack flow is as follows:

1. **Infection:** Attackers infect devices with different malware. The infected devices become sources of the future attack, bots of the botnet. There can be diverse sources for the infection such as fishing, drive-by downloads, vulnerability exploits, etc.
2. **Coordination:** Once the botnet is composed of a high enough number of participants, the botmaster sends commands to ensure the participation of the different bots. This is typically run by servers such as C&C, where the cybercriminal can receive all kinds of data from the bot, in this case usually this is limited to acknowledgement of participation.
3. **Attack Launch:** Simultaneously sending petitions from all edges of the botnet. Via the C&C server, for example, the botmaster commands its zombies to send the specific attack type (SYN flood, DNS amplification…). At the onset, the bots generate traffic to exceed the targets processing capacity.

4. **Overload:** After a small period, the resources are completely saturated provoking a disruption or shut down of service.

Similarly to DoS attacks, it can last from minutes to days causing major outbreak and issues for the victims.

2.5.3 Variants

DoS & DDoS are divided into a vast and complex cluster of different attacks based on the system vulnerability they exploit. The attacks can vary in complexity and method. On this topic there will be an overview of some of the most prevalent and used ones.

The main purpose of **Slowloris** attacks are web servers. It was designed to exploit vulnerabilities on **layer 7** of the OSI model [64]. Used by Anonymous, Iranian government or Killnet as examples, Slowloris opens vast amounts of HTTP connections without finalizing the petition process, at the lowest possible rate. This saturates the servers waiting queue without processing real traffic provoking the damage [65] [64].

It operates in the following manner: The attacker sends **HTTP headers** with missing information, **maintaining** the **connection** opened for the largest possible period. This consumes server resources limiting the ones that can be allocated to well-intended traffic for petitions that never materialize [64].

The vulnerability exploited is in the HTTP server's protocols where no connection limits are stated, or no partial connection management is done. This includes a wide variety of web servers such as previous versions of Apache, Websense, Verizon or versions prior to IIS 6.0 Some of the most notorious Slowloris attacks are the 2009 Iranian elections protests [65] or the Anonymous vs Scientology 2008 attack.

Ping of Deaths (PoD) objective is to **exploit** IP management in **ICMP packages**, crashing the victim's devices. The attacker sends ICMP messages with malformed structures or contents, typically surpassing the size limit. This exchange disrupts the protocol, corrupting the memory of the system and inducing major failures.

Its mechanism is simple; the attacker sends **oversized ICMP Echo Requests**. When reaching the target, it cannot properly process the packages provoking blocks or reboots. The ICMP packets exceed the appropriate size and are therefore divided by the attacker into malformed fragments. When the vulnerable target tries to resample the fragments into the original packet it suffers buffer overflows that lead to the aforementioned results.

PoD dates to the 1990s where the original version of the exploitation was fast patched. It resurfaced with IPv6 and with TCP/IP Stack where vulnerabilities in the Windows OS (vista 7,8 and Server 2008/2012) version where discovered. The main problem that surfaced was the IP protocol not being able to appropriately filter the package by sizes, not being able to detect malformed versions. Some of the most notorious PoD attacks are the Windows 95/NT Ping of Death in1997, the CVE-2015-5366 (Linux Kernel PoD Variant) in2015 or the CVE-2020-16898 ("Bad Neighbour") in 2020.

SYN floods attacks try to **saturate** servers' **connectivity**. They exploit **layer 4s TCP vulnerabilities** in the handshake process. The attacker sends massive amounts of Synchronization packages to the server **without** answering the **SYN-ACK** response, leaving open and incomplete communication establishment intents.

As shown in the figure 6, the attacker first sends **SYN petitions** to the server trying to stablish a TCP link. The server then continues the protocol by sending responses, allocating resources to the client's petition. When numerous petitions are made the server is unable to start new handshake processes and as such new connections.

This vulnerability is present in systems that do not implement packet filtering or have no limit for how many incomplete SYN connections to store. Notable accounts of this attack are Panix in 1996, the Mafiaboy DDoS attacks in the year 2000 and the GitHub DDoS attack in 2018.

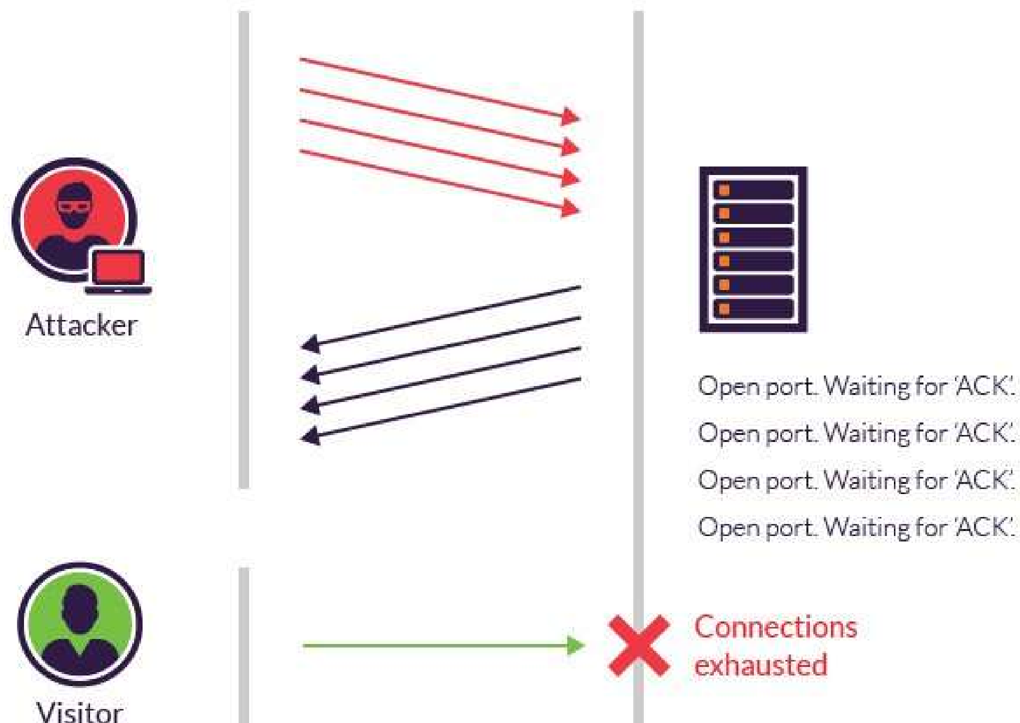

Figure 6. SYN Flood illustration [66]

UDP floods try to exploit the protocols server side when answering **UDP messages** directed to one of its **ports**. Its objective is to saturate the service and **consume** as much **bandwidth** as possible to saturate and delay the response or impair a server.

Attackers send numerous UDP packages directed **randomly** to all server **ports**. When the server tries to answer all the petitions or process all the packages it becomes saturated. Similarly to previously mentioned attacks it overloads the system with **ICMP unanswered packages** when the port it tries to reach offers no service and therefore is inactive.

The flow is simple; the attacker falsifies the origin address of the packages and sends them randomly to a server's ports. When no service is offered the server answers with an ICMP packet, consuming resources. All of these procedures are done massively with the intent to saturate and collapse the victims' network.

Solutions for this attack are UDP traffic filtering or rate-limiting network traffic. Some of the biggest examples of this attack are Memcached DDoS Attack in 2018, Spamhaus DDoS Attack in 2013 or Dyn DDoS Attack in 2016.

There are over 100 types of DoS attacks if considering variations and subtypes. To highlight some more that warrant attention:

- **DNS Amplification:**
    - The objective is to **increase** public **DNS traffic** by sending small inquiries with false origin address so that the DNS server floods the objective with long answers.
    - This exploits public DNS wrongly configured to answer inquiries from any IP address.

  - Some examples are the Dyn attack in 2016 or the Spamhaus attack in 2013.

- **HTTP Flood:**
  - The objective is to **exhaust** the **server's resources** by sending a high volume of seemingly legitimate HTTP requests (e.g., GET or POST), often from multiple sources.
  - This exploits application layer vulnerabilities by mimicking regular traffic, making it difficult to detect.

- **Smurf attack:**
  - The objective is to **overwhelm** the **target** system with ICMP Echo Reply packets by tricking a network into sending mass replies to the victim's IP address.
  - This exploits networks that allow IP-directed broadcasts and hosts that respond to ICMP requests sent to broadcast addresses.
  - Some examples are the University of Minesota attack in 1998 or the ISP Disruptions in 1998.

2.5.4 Attack Classification

Attacks can be divided in several types according to the target layer and the techniques utilized:

- **Volumetric Attacks:**
  - Aim to saturate a target BW by sending vast amounts of traffic.
  - Some examples are the UDP flood or the ICMP flood.
- **Protocol Attacks:**
  - This attack category intends to exploit protocol vulnerabilities and weaknesses to exhaust the victims' resources.
  - SYN flood or PoD are notorious examples.
- **Application-Layer attacks:**
  - They target specific services or applications. They are characterized by more sophisticated and specific structures aimed to exploit vulnerabilities on the application layer.
  - HTTP floods are an ideal example of this attack.

2.5.5 Key Components

DoS/DDoS attacks have key characteristics that compose them and define their structure, traceability and mitigation:

- **Scalability:** Capacity of the attack to adapt to traffic demands, increasing its petitions from Mbps to Gbps when demanded.
- **Traceability:** Capacity of traceback of the attack. Most attacks use spoofing or botnets distributed between nations, making it complex to find out their origin. Technics such as packet marking or backscatter analysis as well as logging advance systems in routers/firewalls are used to discover the perpetrators of the attack.
- **Mitigation Capability**: Capacity of a system to withstand and mitigate an attack, preferably without affecting the legitimate traffic. Key factors are the early detection, automated answers (firewalls, filtering, rate-limiting…) or traffic load balancing.

Under these characteristics, the severity of an attack is assessed, withstood and mitigated. Understanding them is key for organizations to develop appropriate defence strategies that can react to plausible threats as well as prevent future ones.

2.5.6 Mitigation Metrics

DoS/DDoS attack mitigation is a complex and constantly growing endeavour. The different methods employed offer a wide range of effectiveness, measured in different categories:

- **Detection Rate (DR):**
    - Measures the ratio of true positives (TP) in relation to false ones (FP) to reflect the real capability of the system to identify threats.
    - Typical values are over 90%.
    - Realistically, it is impossible to achieve a 100% based on the nature of the attacks DR so services that aim for over 95% are considered excellent.
    - The higher the DR the better the system is at identifying attacks.
- **Accuracy (ACC):**
    - Measurement of TP and true negatives (TN) by total amount of samples in a system.
    - It reflects the percentage of total correct classifications.
    - Typical accuracy levels are of over an 85%.
    - Variations may occur when higher traffic is processed but over a 90% is considered trust-worthy for a detection system.
- **False Positive Rate (FPR):**
    - Percentage of traffic incorrectly classified as malicious.
    - The lesser the better as when traffic is classified as harmful or suspicious it might be suppressed, filtered or dropped by the network
    - Less than 5% is typical for this category.
    - Ideal systems aim to obtain a less than 1% FPR for best security although in some systems that prioritize speed, a FPR of around a 5% might be accepted.
- **Detection Latency or Response Time:**
    - Latency aims to be as short as possible, reducing the damage caused by the attack.
    - Businesses aim for latencies of under 100 ms in critical systems and up to 200 ms in the rest.

2.6 Modern Security Approach

Historically, mitigation techniques have been based on heuristic norms to filter traffic by known patterns or the creation of static Blacklists to block suspicious IPs. These methods provide **limitations** due to high false negative (FN) results, new attacks and complexity on rule modification.

On top of this, the use of exclusively **static Blacklists** quickly becomes **inefficient** against attackers that constantly modify their IP address or utilize identity-theft-based techniques. In dynamic environments like 5G networks or cloud services, where legitimate and malicious traffic can share similar characteristics, it becomes **critical** to adopt more adaptive and automated approaches, such as using artificial intelligence or machine learning to detect anomalous patterns and respond in real time.

A combination of **dynamic Blacklists** and **real-time detection techniques** might offer a promising approach to addressing these challenges. Dynamic Blacklists can adapt quickly to emerging threats by updating suspicious IPs and patterns on the fly, while real-time detection systems provide immediate analysis and response to ongoing attacks. Together, they complement each other effectively as dynamic Blacklists help prevent repeated attacks from known malicious sources, and real-time detection identifies novel or evolving threats that static methods

might miss. This integrated strategy enhances overall security by reducing false negatives and improving the agility of defence mechanisms.

2.6.1 Load Balancing

The principle of load balancing is to distribute the traffic between various servers, preventing saturation of the entire system. Redundance is beneficial when servers fall as the system can maintain providing services with close to no felt impact.

It provides resilience and ensures disponible services. Disadvantages are big attacks or targeted ones that can still bring down the service and the increased cost of all services.

2.6.2 Rate Limiting

Limiting the petitions a server can accept from a specific source in a **timeframe** can provide mitigation to SYN or HTTP Flood attacks. It reduces servers traffic load and limits attacks effectiveness. On the hindside it might affect non-malicious traffic as it can prevent good-intended client's access to the service.

2.6.3 Network-level Filtering.

Routers and switches are employed to detect malicious traffics patterns. It detects certain attacks and blocks them before they can access the network.

This reduces prematurely server loads and allows for network-level mitigation. It can also be very costly and require complex algorithms and configurations.

2.6.4 Traffic Filtering

Firewalls and Intrusion Detection Systems (IDS) are used for detecting and blocking malicious traffics. IDS are some of the most well-known and traditional systems. These systems inspect all entering traffic, interfering with the petitions identified as malicious.

It directly blocks undesired traffic but can conflict with other system or user functions. It can also be non-ideal for big scale attacks, where enormous amounts of traffic have to be analysed. In these cases, not only they might get saturated but also miss important traffic, reducing its effectiveness.

2.6.5 Intrusion Prevention IPS

Intrusion Prevention Systems (IPS) are capable of **detecting** and **blocking** suspicious activities in an active manner. They utilize hash signatures of known attacks and compare them with hash of intended traffic. It is one of the **fastest** detection systems, blocking attacks without the need of human intervention [67]. It is also limited to the hash signature database, losing all of its effectiveness against not known or new attacks [68].

Despite these limitations, IPS continue to be an **essential** tool in perimetral defence. This is especially true when combined with more advances techniques like behavioural analysis or IA pattern detection systems. Combining techniques seems to surpass dependency on signatures, improving response capacity against emerging threats and day zero attacks.

2.6.6 Machine Learning Based Techniques

The use of **Machine Learning algorithms** provide an improved detection system as it can offer better rates when attacks do not follow linear patterns. ML algorithms can detect more complex patterns.

There are several interesting, supervised learning algorithms that can be implemented such as SVM or KNN. The algorithm can be trained with different datasets such as **KDD99, NSL-KDD, CIC-IDS2017** or **BoT-IoT**. This offers great **potential** if adjusted properly but can have problems adapting to real-world traffic.

Enterprises can opt to assemble their own datasets with real traffic to assess integration issues.

2.6.7 Deep Learning & Neural Networks

Deep learning, particularly CNNs, can offer conspicuous detection capability. They can learn temporal patterns and spatial ones to identify anomalous traffic. These algorithms are particularly interesting for algorithms that evolve over time and large, complex datasets.

Thay offer better capacity for complexity and more potential for precise detection. They also consume higher amounts of resources and require bigger datasets for training.

2.6.8 Distributed Systems

Distributed systems can be used to **share** information between **nodes**, communicating important data regarding the attacks, from detection to prevention. A collaborative approach can optimise resource usage and simplify detection of threats in real time. This system can also distribute the workflow, easing the processing and workload.

They can offer **worse latencies** than other systems and can have problems of complexity of implementation and a harder scalability. Some of these issues can be helped with blockchain integration.

2.6.9 SDN

SDN can be vulnerable to attacks directed to the SDN controller or network congestion. To detect these patterns there are:

Detection Approaches:

- Intelligent Controllers: SDN controllers can leverage machine learning techniques to identify anomalous patterns and dynamically distribute traffic across the network, helping mitigate DoS attacks.
- Data Flow Monitoring: Analysing traffic flows between SDN devices using real-time monitoring tools and techniques such as correlation analysis and anomaly detection helps identify attack patterns like link saturation or disruption of specific data flows.
- Distributed Defence: In SDN networks, distributed defence mechanisms can be deployed at various points of the network to detect DoS attacks without relying on a single point of failure.

2.7 Challenges

DoS Attacks propose multiple challenges from technical to organizational. In this project there is no definite proposal to eradicate these attacks but there is an analysis of different partial solutions and open issues through the combination of automated learning techniques and blockchain technology.

2.7.1 Reliability and traceability in Blacklist management

One of the main issues with traditional Blacklists is their centralized nature. This can induce multiple vulnerabilities such as data manipulation, availability issues and transparency concerns. In this context, this project pretends to present the use of blockchain as an alternative for secure maintenance of this information. Blockchain is already in use by big companies as a ledger of information and it can warrant the system a decentralized, trustworthy, public and verifiable data storage. This can ease decentralized management and

improve trust on malicious event detection, especially the need of more transparent systems, resilient to manipulation.

2.7.2 Entity collaboration

Sharing information about cyberattacks between organizations is often limited by trust, privacy, or regulatory concerns. By using a blockchain infrastructure, this work enables different actors to access recorded malicious events without needing to trust a central authority. While data anonymization and advanced privacy mechanisms are not addressed, it establishes a foundation for open collaboration among stakeholders.

2.7.3 Automated integration between detection and logging

In many environments, detection systems and reporting or incident management systems operate separately, hindering a rapid response to threats. This project proposes an architecture in which the detection module, once it classifies traffic as malicious, automatically generates a record on the blockchain. This reduces the gap between analysis and action, promoting more seamless integration between system layers.

2.7.4 Data management and preprocessing for model training

The development of intelligent detection systems is hampered by the scarcity of representative and well-labelled data. In this work, a realistic dataset was selected, and extensive preprocessing of relevant features was performed to maximize model performance. Although the solution does not fully address the problem of accessing traffic data in real-world environments, it does contribute to practical knowledge on how to build effective and reproducible training pipelines.

3 Analysis, design, and solution implementation

3.1 Requirements

System requirements have been divided into two categories: functional and non-functional. The former describes the basic capabilities the solution must offer to meet its objectives. The latter establish conditions for quality, performance, security, and ease of use or maintenance.

3.1.1 Functional Requirements

FR1. The system must be able to capture network traffic in real time or from previously recorded files.

FR2. The system must process traffic characteristics and transform them into a format suitable for analysis.

FR3. The system must classify traffic as benign or malicious using a pre-trained learning model.

FR4. The system must allow querying previously recorded events from the blockchain.

FR5. The system must be easily integrated with other network tools or platforms through defined interfaces (e.g., REST APIs or automated scripts).

FR6. The system must maintain a warning counter per IP and block sources that exceed a predefined threshold.

FR7. The system must log each detected attack to a blockchain in a tamper-proof and verifiable way.

FR8. The system must expose a RESTful interface allowing actions such as retrieving, inserting, and deleting attack records.

FR9. The system must reject incoming API requests from IP addresses that have been Blacklisted.

FR10. The system must support manually submitting attacks via the frontend or API for testing or external integration.

FR11. The system must allow automatic periodic updates from the blockchain to ensure Blacklist freshness.

FR12. The system must enable the simulation of benign and malicious traffic for testing purposes.

FR13. The system must generate and persist log files including timestamps, component source, and severity levels.

3.1.2 Non- Functional Requirements

NFR1. The system must respond to traffic classification in less than 2 seconds per sample.

NFR2. The solution must guarantee the persistence and immutability of logged events through the use of blockchain technology.

NFR3. The system design must be modular, allowing key components such as the machine learning model or the smart contract to be updated independently, without requiring major changes to the overall architecture.

NFR4. The solution must be adequately documented, including usage instructions, deployment steps, system structure, and interface descriptions, to support long-term usability and expansion.

NFR5. The system must gracefully handle network-related errors, temporary disconnections from the blockchain, or classification module failures, ensuring that stability and basic functionality are preserved.

NFR6. The system must ensure concurrent-safe access to shared data structures such as the Blacklist using appropriate synchronization mechanisms.

NFR7. All blockchain operations must support traceability, allowing each transaction to be verifiably linked to a specific IP address, attack type, and timestamp.

NFR8. The system must produce centralized logs (e.g., packets.log, Blacklist.log) including timestamped records of key events for auditing and debugging purposes.

NFR9. The user interface must be intuitive and responsive, enabling Blacklist management without requiring deep technical knowledge.

NFR10. The system must allow runtime configuration of key operational parameters such as the warning threshold or the automatic update interval.

3.2 Design

The design section intends to present the **internal structure** of the system. It addresses its **functional analysis**, **modular** component **design** and some of the most **relevant aspects** of its implementation. The main approach has been a **modular architecture**, trying to allow for the most **separation** of functionalities, from traffic capture to persistence logic processing. Various **UML diagrams** such as component, class or sequence diagrams try to graphically represent the solution, facilitating the understanding of the various interactions between system elements.

3.2.1 General Architecture

The figure 7 shows the **systems architecture**. This consists of two main modules that act independently, interacting through the public VeChain blockchain. There also is a third external module that compliments the design, simulating traffic and as so validating the detection.

The module structure is as follows:

- **Malicious traffic detection module:** Known throughout the document as DoSDetector, this module is responsible of capturing real-time traffic, extracting its most relevant features and applying a supervised learning model to classify it. It then records the positive attacks as new entries on the public blockchain. It is divided in three main modules:
  - **Traffic interception and processing**, located in the metrics.py file. Network packets are intercepted in real time through the use of tools such as Scapy, then it calculates a set of pre-decided metrics from each packet flow and sends them to the classificatory node.
  - **Traffic classification**, implemented in the predictor.py file. It receives the extracted metrics transformed into a vector. This vector is later analysed through a pre-trained, supervised model. The model returns a label associated with the traffic type. This assessment decides whether a warning is issued towards the IP address associated with the vector, whether to clean the warning counter for such IP or whether to block it.
  - **Registration and consultation in blockchain,** located in the blacklist.py file. When traffic is labelled as malicious and the warning threshold is surpassed, the event, in a JSON form, is logged on the blockchain. It utilizes the blockchain folder´s scripts to communicate the transactions or petitions. These logs are part of a decentralized and verifiable Blacklist.
- **Blacklist query and management server module:** This module, called simply server through the project files, implements a service for periodically querying the smart contract deployed on the VeChain test-net. It updates a local copy of the Blacklist, storing malicious events information. This registry can then be

used externally by applications or monitoring systems. This node main functionalities are to act as a management server, exposing a REST interface and an interactive web dashboard that allows:

- Consulting the current status of the Blacklist.
- Manually logging new events.
- Deleting or updating existing entries.
- Setting automatic update intervals.

All of this is done while maintaining consistency between the local status and the information stored in the blockchain.

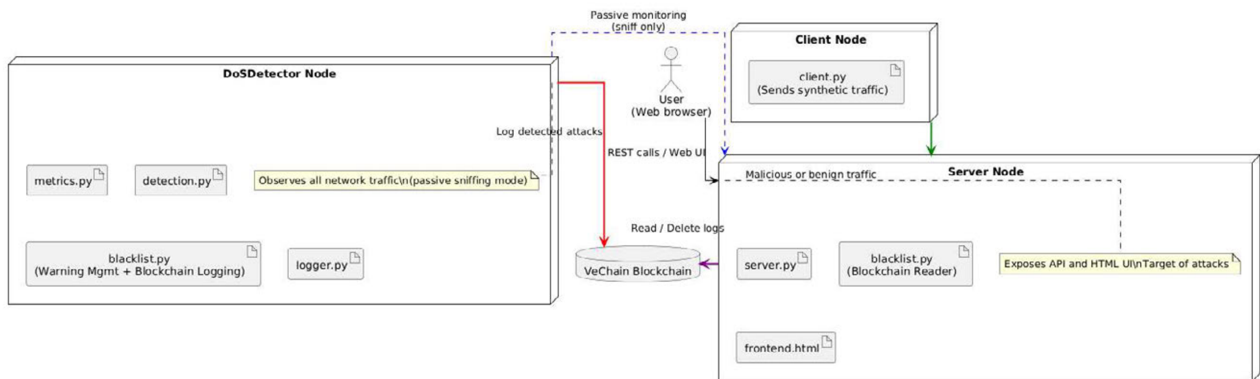

Figure 7. General Architecture Scheme

The third **non-architecture** related nodule is the traffic-generating **client**. This nodule, encapsulated in the client folder of the project, is responsible of generating various traffic types. It is capable of **simulating** benign or DoS attacks (e.g., SYN Flood, UDP Flood, or HULK). Its main function is the **validation** of the entire system performance by ensuring that attacks can be detected and logged efficiently. This component is not part of the production architecture but is essential during the evaluation and implementation phases.

The modular **separation** facilitates the threat detection logic to be differentiated from the distributed persistence logic and the query logic, facilitating system scalability, component reuse, and integration with future external tools. It also supports a progressive **evolution** of the system without affecting its stability, allowing the incorporation of new types of attacks, more sophisticated classification models, or other blockchains without the need to redesign the entire solution.

3.2.2 Blockchain Selection

One of the most crucial aspects of this project was the blockchain selection. The transition from HLF to VeChain has been significant for the major architectural decisions made throughout the projects design phases. The first versions were made with HLF, a permission-based network but it was overruled due to the following reasons:

1. Deployment: VeChain has a public testnet and REST-API based tools. This simplifies the configuration, eliminating the design of an entire network structure from scratch (nodes, organizations, orderers, certificates…)
2. Scalability: VeChain is a public blockchain, meaning it guarantees availability without the need of identity management.
3. Ethereum Virtual Machine: VeChain is compatible with EVM, enabling the possibility of Solidity contacts. This facilitates integration with Eth based tools.
4. Maintenance: HLF requires nodes, channels and Chaincodes to be supported on the network deployers and actors. VeChain on the other hand allows a lightweight architecture where all the heavy loads are taken by organisms external to the scope of this project.

The table presents a comparison between both technologies.

Table 2. Comparison. HLF vs VeChain

Feature	Hyperledger Fabric	VeChain	Project Impact
Network Type	Permissioned	Public and decentralized	VeChain improves transparency and openness
Consensus	Modular (Raft/Kafka)	Proof of Authority (PoA)	VeChain reduces cost and latency
Infrastructure	Complex, requires Docker, TLS, certificates	Lightweight, API-accessible	Easier deployment with VeChain
Development	Specialized Chaincode (Go/Node)	Standard Solidity contracts	Faster and simpler integration
Public Scalability	Limited	Global public access	Broader access with VeChain
Operational Costs	High (multi-container setup, persistent storage)	Low (testnet faucet, no infra management)	More suitable for frequent use
Ecosystem Focus	Corporate/private networks	Supply chain and traceability	Strong fit with Blacklist use case
Write latency (Tx)	2.3–3.1 s (unoptimized on HDD RAID 5)	1.5–2.1 s (confirmed on public testnet)	Faster persistence with VeChain
Read latency (Tx lookup)	~200 ms (local ledger read)	300–450 ms (via RPC endpoint)	Acceptable read delay with improved accessibility
Python Integration	Requires SDK, gRPC, or REST gateway	Simple subprocess call to Node.js script	Lower code complexity using VeChain

As shown in figure 8; in the initial phase of the project, Hyperledger Fabric was used with custom Chaincode to manage the Blacklist, leveraging its permissioned architecture that facilitates control and security within private networks. However, for the final implementation, the decision was made to migrate to VeChain and its standard Ethereum-based smart contracts, which allowed for easier integration, greater public accessibility, and reduced operating costs, better adapting to the system's scalability and transparency goals.

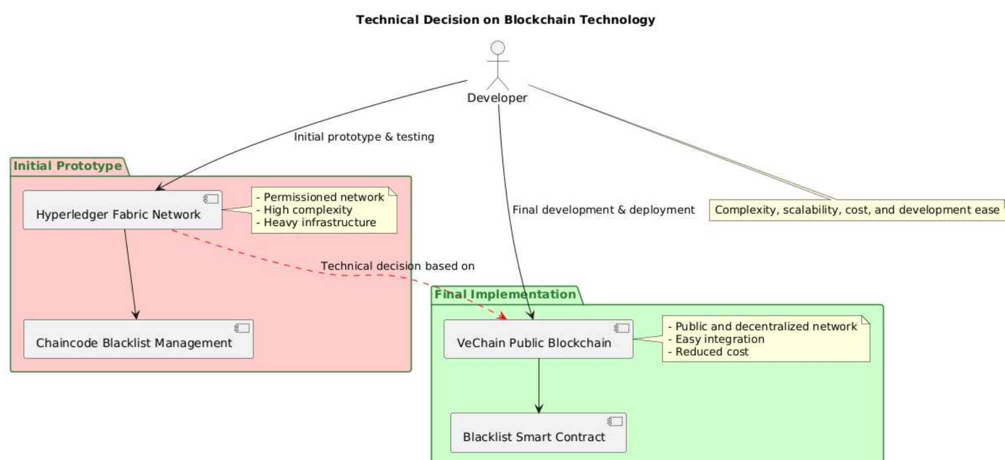

Figure 8. Blockchain Technical Selection

3.2.3 Machine Learning Models Overview

A supervised learning model is used the malicious traffic detection part to classify network traffic. This model is utilized by the DoSDetector module by sending the collected data of each packet flow. This data is composed of a wide variety of metrics extracted directly from the network. The model is capable of classifying anomalous patterns indicating the possibility of an attack.

Among the algorithms implemented and evaluated for this task there are several classic and effective models for classification problems, such as K-Nearest Neighbours (KNN) and Random Forest. After an exhaustive process of comparison and testing with different data sets, it was decided that the user would be able to determine which model to integrate into the system. Classifiers are great models for this task for several key reasons:

- **Ability to handle large volumes of data:** Supervised learning algorithms, thanks to their nature based on direct sample comparison, can scale relatively well in scenarios where the amount of data is considerable, provided that appropriate data structures and optimizations are used.
- **Suitability for live detection scenarios:** Most supervised learning algorithms are particularly effective in scenarios where no retraining is needed during inference. As implemented in this project, the pre-trained model operates on real-time metrics with minimal preprocessing, which aligns well with the lightweight, modular design of the system.
- **High detection accuracy:** The models demonstrated outstanding performance in terms of attack detection rate as shown in figure 9, often outperforming other more complex models. This translates into a greater ability to correctly identify malicious events without generating too many false positives.

The integration of these models into the system enables proactive and accurate detection, significantly increasing the system's ability to identify emerging attacks and adapt to new threats. Furthermore, the system's architecture is designed in a modular manner, facilitating the updating and replacement of machine learning models, thus promoting scalability and continuous improvement. This is especially relevant in a cybersecurity context, where attackers' tactics are constantly evolving, and the system must be kept up to date with the latest techniques and data to maintain its effectiveness.

```
[INFO] Cargando y procesando logs...
[INFO] Clases originales: {2: 881, 1: 810, 0: 789, 3: 703, 4: 697}
[INFO] Aplicando SMOTE para balancear clases...
[INFO] Entrenando modelo KNN...
[INFO] Generando reporte de clasificación...
              precision    recall  f1-score      support
BENIGN         0.992647  1.000000  0.996310   270.000000
HULK           0.982699  0.982699  0.982699   289.000000
SYNFLOOD       0.984375  1.000000  0.992126   252.000000
UDPFLOOD       1.000000  0.973180  0.986408   261.000000
POSTFLOOD      0.992032  0.996000  0.994012   250.000000
accuracy       0.990166  0.990166  0.990166     0.990166
macro avg      0.990351  0.990376  0.990311  1322.000000
weighted avg   0.990231  0.990166  0.990147  1322.000000
[INFO] Reporte guardado en: classification_report_knn.csv
[INFO] Modelo guardado en: knn_model.pkl
[INFO] Precisión media (cross-validation): 0.9909
```

Figure 9. KNN Model statistical recollection

From a technical point of view, the model development and integration process followed the following main phases:

1. **Feature Extraction and Preprocessing:** Relevant metrics extracted from network traffic were defined and selected, such as packet rates, inter-arrival times, packet lengths, TCP flags, and others. These features form the basis for the models to differentiate between legitimate and malicious traffic.
2. **Supervised model training:** Using labelled datasets that included examples of benign traffic and traffic under different types of DoS attacks, multiple models were trained. Cross-validation techniques were applied to evaluate the algorithms' performance and avoid overfitting problems.
3. **Performance evaluation:** Multiple metrics were collected, including accuracy, precision, recall, F1-score, and inference time. These were used to benchmark algorithms and select the one offering the best trade-off between speed and detection power.
4. **Integration and deployment**: The selected model was exported as a .pkl file and embedded into the DoSDetector module. At runtime, the module receives metric vectors, performs inference, and, when malicious behaviour is detected, updates internal counters and potentially triggers blockchain logging.

To better organize the documentation and prevent for this section to be overloaded, all information regarding configuration, experimentation or validation of the model as well as quantitative measures are included in **appendix I**.

To sum up, the use of a supervised learning model offers an **efficient**, **fast** and adaptable **solution** for early DoS attacks detection as it **minimizes false positives** and establishes a **proactive defence** without compromising the operational performance of the system.

3.2.4 Component Design

Each component is constructed as a standalone module, based on independence. Nonetheless each component is coordinated through VeChain, maximizing scalability, maintainability and ease of update. This approach aligns with software engineering principles.

The system is divided into three main modules as previously stated. Each has several well-defined responsibilities and specific entry points:

- **DoSDetector:** it is composed of several files that ensure its previously defined objectives:
  - Its main script, *metrics.py,* captures packets using Scapy and extracts relevant features.
  - The *detection.py* component contains the classification engine based on supervised models, highlighting the use of supervised algorithms to label traffic and detect DoS attacks.
  - When an attack is detected, the *logger.py* module records the event locally.
  - The *Blacklist.py* interface, invokes a Node.js script (sendAttackLog.cjs) to store the event on the VeChain blockchain. The local management of warning counters and Blacklist state is handled by this *Blacklist.py* as well, which tracks the number of warnings per IP and triggers a blockchain registration upon exceeding the defined threshold.
  - Machine learning models are organized into separate folders within *DoSDetector/models/,* allowing comparisons between different algorithms and making it easy to update or replace models without impacting the rest of the detection pipeline.
- **Server:** This module provides the REST API implemented through:
  - *server.py* is responsible of most of the systems logic, using the blacklist.py file to respond to queries about the current Blacklist. It allows operations such as inserting, deleting or clearing entries. Internally, it also uses its own common JavaScript scripts (e.g., *getAttack.cjs, deleteAttack.cjs*) to interact with smart contracts deployed on the VeChain blockchain. Additionally, the server embeds a fully functional HTML frontend (frontend.html) to allow non-technical users to visualize and manage Blacklist entries.

- **Client:** Standalone, this module contains an only file labelled *client.py*. It generates malicious traffic directed at an url. Its main use is as support for simulating events, malicious and benign. This permits the system to be evaluated on performance and detection response under realistic conditions.

The first two modules count with a shared *Blacklist/* folder. It contains the aforementioned scripts for blockchain management. This facilitates the integration between different runtime environments, allowing Python to handle core logic while delegating blockchain-related tasks to be executed via subprocess.

The system architecture adheres to various well-established software engineering principles:

- **Modularity:** Each module has a clear, differentiated and unique responsibility, which simplifies testing, maintenance, and future evolution.
- **Reusability:** The scripts located in the *Blacklist/* folder are shared between the detection and server modules, avoiding code duplication and ensuring consistency in blockchain operations.
- **Extensibility:** Adding new supervised models only requires placing the new model file or folder under *DoSDetector/models/* and updating the AttackDetector configuration, without the need to alter the rest of the architecture.
- **Interoperability:** The hybrid use of Python (for detection and system logic) and JavaScript/TypeScript (for blockchain interaction) provides cross-platform flexibility and ease of integration with existing ecosystems.

In short, the organization based on independent components provide a robust and scalable solution. It offers transparency and decentralized blacklist management via Blockchain technology.

3.2.5 Object Oriented Design

With bases in the modular components defined in the previous section, the system adopts a carefully structured **object-oriented design**. It emphasizes **modularity** and responsibilities **separation**. This approach pretends to be aligned with good software engineering practices and to ensure each functional block to be evolvable independently without damaging interoperability.

The system uses **SOLID** principles to achieve modularity. This ensures a clear separation between responsibilities and encourages the reuse of code. This architecture is meant to facilitate the independent evolution of each component. It also permits the addition of new functionalities without compromising the ones already implemented:

- **Single Responsibility Principle (SRP):** Each module is designed to compel with one function. As an example, the MetricsExtractor (metrics.py) is exclusively in charge of capturing and extracting traffic, whilst AttackDetector (detection.py) makes predictions given any data. This separation allows versatility, enabling maintenance and scalability.
- **Open/Close Principle (OCP):** Each component is open to improvement but closed to modifications. It is possible to introduce new functionalities without modifying the system core. Several examples are found:
  - The detection is adapted to a model with specific attacks but can be improved by just changing the label map and the model itself for increasing the number of detected attacks.
  - The utilized blockchain is VeChain but that can be modified by just changing the common JavaScript files from the node-modules folder, allowing the change of blockchain without modifying the system.
- **Liskov Substitution Principle (LSP):** Although the system does not make extensive use of class hierarchies, behavioural contracts between components are respected. Where inheritance or

composition has been used (for example, in HTTP handlers), derived classes are guaranteed to be able to replace the originals without generating unexpected side effects.

- **Interface Segregation Principle (ISP):** Components interact with each other through minimal, well-defined interfaces. For example, the HTTP server calls functions like log_attack(ip, type) without knowing or depending on low-level details, such as calls to external processes or Node.js scripts.
- **Dependency Inversion Principle (DIP):** High level modules are not dependent of concrete implementation but of abstract. This can be reflected in decisions such as loggers, detectors, or Blacklist managers inside constructors of classes, facilitating decoupling. No formal interface was designed but the architecture is built to be flexible and easily integrated.

The use of SOLID has allowed for a clear **task separation**, encouraging **reutilization**, easy **testing** and **independent evolution** for each component. Because of this implementation, new functionalities can be easily incorporated without compromising the systems stability. Examples could be more complex ML models, more advances access politics for the system, a variation in the blockchain or smart contract improvements.

The system is organized into two main modules: the DoSDetector module and the Blacklist Server module, as well as an auxiliary module dedicated to traffic simulation. Each one includes classes that encapsulate specific functionalities and collaborate with each other.

The DoSDetector module is responsible for capturing network traffic, extracting metrics, detecting attacks, and managing the local Blacklist. It also persistently logs detected events. Its main classes are:

- **Metric Extractor:**
  - It captures and processes network packets in real time.
  - Its key responsibilities are:
    - The use libraries such as Scapy to capture traffic.
    - Filtering and parsing packets (TCP/UDP/IP).
    - Extracting relevant features (flags, size, timing, etc.).
    - Generating data structures for detection.
- **AttackDetector:**
  - It classifies flows as benign or malicious using supervised models.
  - Its key responsibilities are:
    - To load pre-trained models (joblib,. pkl).
    - To preprocess metric vectors.
    - To apply the model and obtain predictions.
    - To trigger Blacklist updates and notify the logger.
- **Blacklist:**
  - It manages IP warnings and blocks within the detector.
  - Key responsibilities of this class are:
    - Loging IP warnings.
    - Deciding when an IP should be blocked.
    - Invoking the sendAttackLog.cjs script to log attacks on VeChain.
- **Logger:**
  - It records system events and decisions in .log files.
  - Its main responsibilities are:
    - To write logs with levels (INFO, WARNING, ERROR).
    - To log predictions, IPs, attack type, and timestamp.
    - To facilitate traceability for later analysis.

The core classes of the DoSDetector module interact in a clear and **sequential** manner: the Sniffer captures packets and generates flow-level metrics, which are sent to the AttackDetector, which is responsible for classifying the traffic. If malicious behaviour is identified, the AttackDetector consults and updates the local Blacklist, which keeps track of warnings and blocked IP addresses. When an attack is confirmed, the Blacklist triggers event logging, and the Logger is responsible for persistently storing it with all relevant metadata.

The second module (Blacklist Server, REST + frontend) module operates independently of the detector, offering a REST API for managing the Blacklist, as well as a web interface for users. Its main classes are:

- **Blacklist (Server)**
  - It stores and synchronizes the Blacklist with the blockchain.
  - Key responsibilities of this class are:
    - To query and update the list in memory.
    - Executing CommonJS scripts (getAttack.cjs, deleteAttack.cjs, etc.).
    - Ensuring consistency with logs in VeChain.
- **Server:**
  - It manages REST requests and serves the web interface.
  - Its main responsibilities are:
    - To provide endpoints (/Blacklist, /log, /delete, etc.).
    - Restricting access from blocked IPs.
    - Starting a thread that periodically updates the Blacklist with the blockchain.
    - Managing clean shutdowns (system signals).

Although not part of the main detection flow, the (Traffic Simulator, client.py) module allows you to **generate traffic** to test and validate the system.

The main function of the traffic simulation class is to generate benign and malicious traffic directed to the server, with the goal of testing the system's detection capabilities. Its responsibilities include generating traffic with configurable patterns (such as SYN Flood attacks, HULK attacks, etc.), switching between different operating modes (benign, slow attack, mass attack, among others), simulating multiple IP addresses, ports, and protocols, as well as evaluating the model's behaviour against false positives and its sensitivity to different attack conditions.

This class interacts indirectly with the main system, as it sends traffic to the network being monitored by the Sniffer, allowing the entire detection flow (from capture to recording in the blockchain) to be validated without the need for manual intervention.

Overall, this object-oriented structure allows for a **robust**, **scalable**, and **flexible system**. Each class fulfils a clear function, maintains its **logic encapsulated**, and **communicates** with others through defined **interfaces**. This organization favours code maintainability and the integration of new models, scripts, or services in the future, without the need to rewrite the underlying architecture.

A visual representation of the class structure is provided in section 3.3 (Implementation), where each module's internal logic is illustrated with UML class diagrams and relevant code snippets.

3.2.6 Operational Flow

Building on the object-oriented design described in the previous section, the system's operational flow reflects a modular execution pipeline where each component performs a specific task ensuring scalability, traceability, and reliability. Below, the full detection and response cycle is described step by step.

The system's operation relies on a distributed architecture in which each node fulfils a specific function but is **synchronized** with the rest through well-defined mechanisms, such as the use of blockchain and shared structures. The overall flow can be understood by dividing the system into two main processes: the detector node flow and the Blacklist server flow, in addition to the malicious traffic flow that functionally connects them.

The detector node functions as a **passive** network traffic analysis tool. Upon startup, the metrics.py module initiates a packet capture process on the selected interface. As traffic flows through the network, packets are stored and processed to group them by source IP. For each flow, statistical metrics are calculated that summarize its behaviour: number of packets, duration, TCP flags, packet sizes, arrival times between packets, among others. These metrics allow the flow to be characterized quantitatively.

Once a flow has accumulated enough data to be analysed, a feature vector is generated and passed to the detection.py module, which loads a pre-trained model to classify the traffic. If the model detects malicious behaviour, the system updates the associated IP's warning list. If the defined threshold is exceeded, the malicious nature of the traffic is considered confirmed, and an event is logged in the local Blacklist.

From that point onward, the detector's Blacklist.py module executes a Node.js script (sendAttackLog.cjs) that logs the attack as a transaction on the VeChain public blockchain. This operation ensures that the event is recorded permanently, immutably, and accessible from any part of the system or by third parties. Finally, this entire sequence, from capture to blockchain logging, is documented in persistent logs using the logger.py module, allowing for auditing, traceability, and forensic analysis.

This complete flow is represented in Figure 10: Operational flow of the detector node.

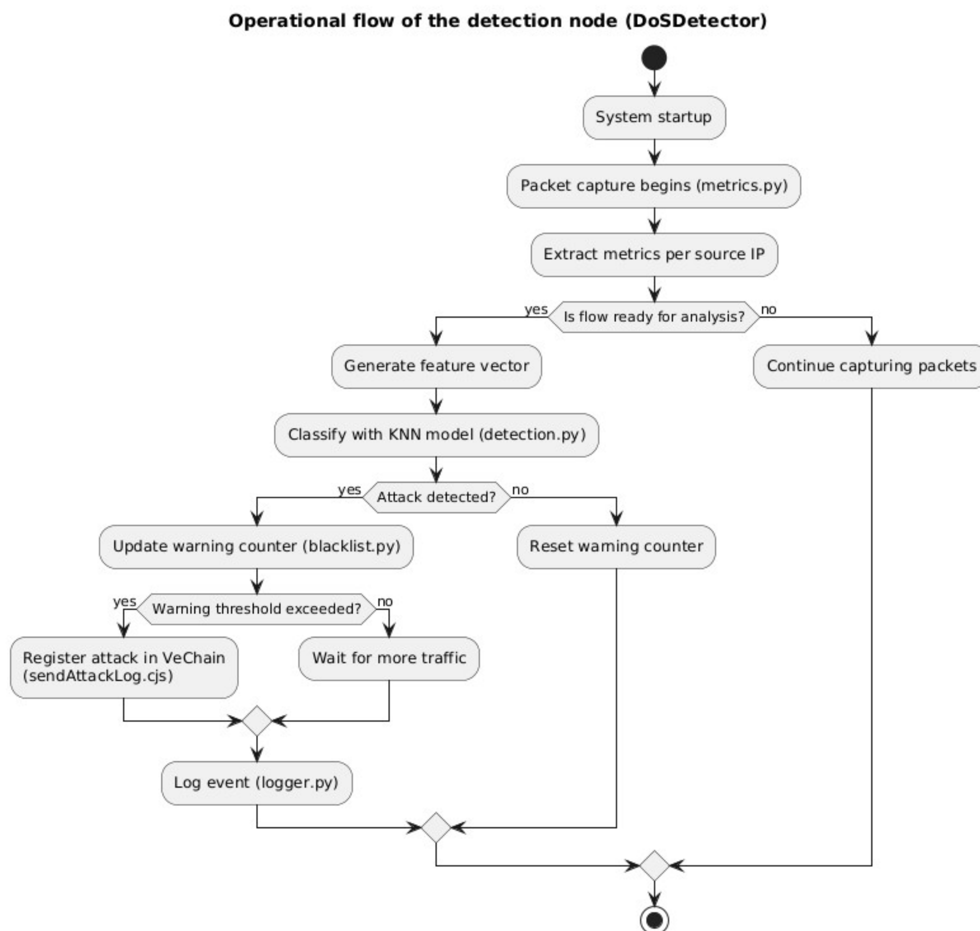

Figure 10. DoSDetector Operational Flow

The server is responsible for exposing the **Blacklist** to users and keeping a **synchronized** version of it. When the server is started, it first synchronizes with the status of the blockchain. Secondly, it exposes a multithread **HTTP server** that listens for requests on a defined port. In addition to serving **REST** requests, the server runs a **background** thread that **periodically** updates the local Blacklist, querying the current status of the smart contract.

When an external client queries the Blacklist via a **GET** request to the /Blacklist endpoint, the server responds with the updated list in **JSON** format. If the client's IP is registered as malicious, the system blocks the request and returns a **403 error**. Additionally, the server allows other operations such as **logging test attacks**, **deleting** specific **entries**, or **emptying** the entire list, either from the **API** or from the **web interface** provided by frontend.html.

All of this logic is designed to operate continuously, capturing system signals (such as Ctrl+C) and ensuring a controlled and clean shutdown, leaving no hanging threads or inconsistencies. The complete server cycle, from startup to controlled shutdown, can be visualized in Figure 11.

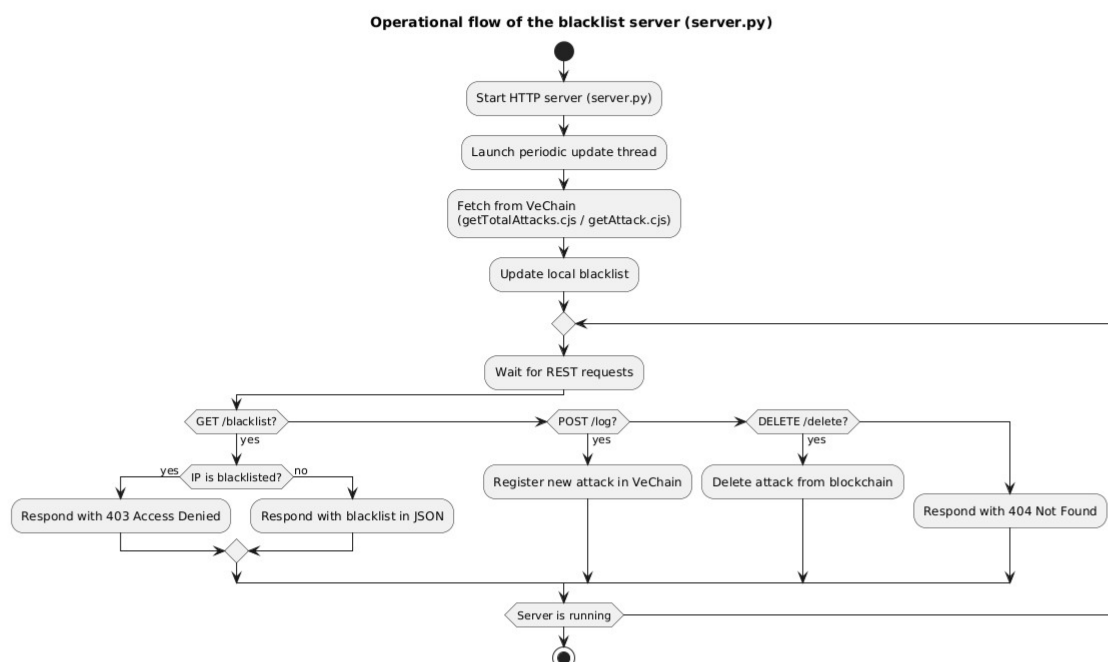

Figure 11. Blacklist Server Operational Flow

The traffic that triggers this system can be generated in a controlled manner using the client.py module, which simulates different types of attacks (SYNFlood, HULK, UDPFlood, etc.), or it can come from real sources in a broader network environment. In both cases, this traffic reaches the server node, which is unaware of its nature. However, the detector, operating in passive mode, intercepts all packets, analyses them, and determines whether the response chain should be activated.

Thus, the flow of malicious traffic passes through the system in three phases:

1. It **reaches** the **server**, **activating** its endpoints or **services**.
2. It is analysed by the **detector**, which **evaluates** it without intervening.
3. If it is **detected** as an attack, it is **recorded** in the local **Blacklist** and then in the blockchain, blocking future connections from that IP.

This process is summarized in Figure 12: Malicious traffic flow and system response.

The modular approach enables an effective threat detection and recording while facilitating maintenance, system integration and enables project evolution. A combination of passive monitoring, automated classification and distributed persistence construct, in the solution, a lightweight yet powerful environment ready for real-world integration and research environments.

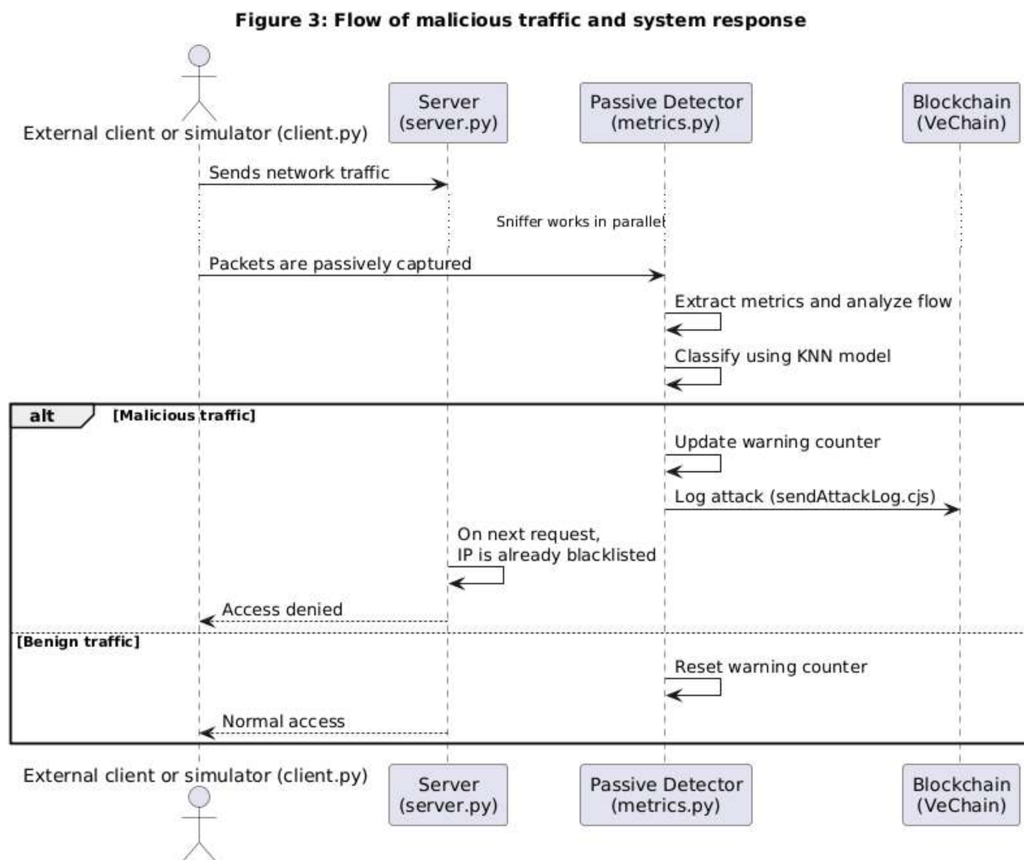

Figure 12. Malicious Traffic & System Flow

3.2.7 Data Model

As the system operates, it relies on a lightweight, well defined, robust data model. The model has to offer consistency, traceability and efficient information handling. This section describes data structures and key data entries within the system.

The data design aims to clearly and efficiently represent the needed information either for malicious data detection or for associating events and flows to specific entities. It ensures consistency by defining communications between local modules and persistent storage in blockchain.

Within the system, three main structures are designed:

1. **Blacklist entries:** Information sent to the blockchain representing malicious events. It is composed by:
    a. **IP Source:** Address associated with malicious actors.
    b. **Attack type:** Differentiates event type.
    c. **Timestamp:** Extracts the precise time when the event is detected and logged.

    These entries are sent from the detector node to the VeChain ledger and from the blockchain they are retrieved to any consulting node.
2. **Traffic metrics:** these metrics quantify the behaviour of observed IP flows. They are used as input vectors for the model. Although not stored in a structured database, the system maintains a log (packets.log) that stores relevant entries for traceability and system validation.

3. **System logs:** Data designed to manage the persistent writing of events. It is stored in log files, such as *blacklist.log*. They capture key system actions, registering them with timestamps and severity indicators. The information stored is useful for auditing, debugging, and performance improvement.

The server maintains a **local copy** of the Blacklist, which it periodically **updates** by querying the smart contract. This synchronization ensures that the responses provided by the API are based on reliable and up-to-date information, without relying exclusively on the detector.

3.2.8 Deployment Scheme

The system was designed to operate under a completely **decentralized** deployment scheme, in which the different modules operate **independently** and **autonomously**, and the only means of synchronizing and sharing information is the VeChain public blockchain. This architecture guarantees scalability, portability, and robustness against individual failures, eliminating direct dependencies between nodes and avoiding single points of failure.

The **DoSDetector** node is deployed **locally** on workstations, servers, or monitoring devices located within the **network** to be protected or monitored. Its function is to passively capture traffic in real time, process packets to extract relevant metrics, and apply pre-trained machine learning models to classify traffic as benign or malicious. If an anomalous pattern is detected, the node **does not communicate** directly with any server or other detectors: it **records** the event directly on the blockchain by executing specific scripts that interact with the smart contract deployed on VeChain. Each detector node operates completely autonomously, without the need for coordination with other nodes.

The **Blacklist server**, for its part, is deployed on a machine accessible via a local or public network, acting as a **query** and **management** point for the **Blacklist**. This module does not directly receive notifications from the detectors; instead, it periodically queries the smart contract to obtain events recorded by any node in the system. The retrieved information is stored in a synchronized local list, which serves as the basis for responding to HTTP requests (for example, from the REST API or the frontend). This separation ensures that the server does not depend on the availability of the detectors and can provide continuous service even in the event of a partial system shutdown.

The **VeChain blockchain** acts as a single common and decentralized **source of information**. All detected attacks are immutably and publicly recorded through smart contract function calls and can be consulted by any authorized entity accessing the network. Thanks to this layer, there is no need to establish direct communication channels between modules: data synchronization occurs implicitly, through shared access to the blockchain.

Under this deployment model, multiple detector nodes can work in parallel independently of networks or environments. No connection between same-type nodes is needed; one node can work without any information regarding the rest of the structure. Similarly, any number of Blacklist server instances can be deployed for redundancy or load balancing. All information flow is channelled exclusively through the blockchain. This provides resilience, auditability, and guaranteed traceability.

Blockchain interactions are carried out **through** scripts developed in **TypeScript** and transformed into **common JavaScript.** These scripts act as clients for the smart contract. These scripts are built upon VeChain official SDKs. Operations are logging querying or deleting attacks. The scripts are executed as subprocesses by the corresponding python file or the persistence logic (deployed on VeChain). This separation reinforces the modularity of the system, facilitating integration of new features without affecting the rest of the system.

Figure 13 shows a graphic visualization of the architecture, depicting its model's physical layout, logical isolation and synchronization flows. These flows can be seen converging exclusively in the blockchain.

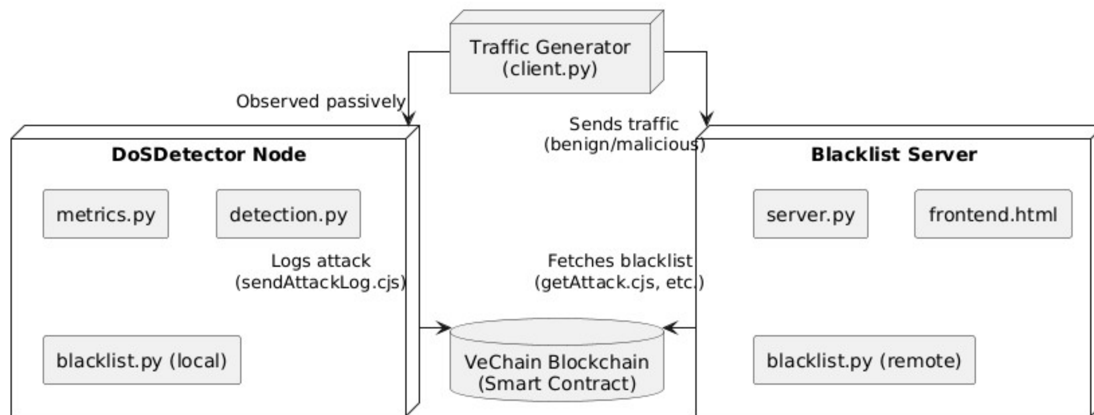

Figure 13. System modular layout

3.3 Solution Implementation

Modular design principles described in previous sections were followed throughout the entire implementation process. The main objective was to comply with all requirements through differentiated components. Each component was developed independently, with particular emphasis on clearly defined responsibilities and interaction mechanisms.

The primary programming language used was python for the main system components and common JavaScript/TypeScript for the blockchain management. The python defined components managed the core logic: traffic capture, analysis, predictions and API management, whilst common JavaScript was used mainly to query the VeChain blockchain. The latter were executed as subprocesses, allowing the detection logic and persistence layer to maintain separate.

The following sections detail the most relevant aspects of the implementation of each of the main modules: the DoSDetector node, the Blacklist management server, and the blockchain-based persistence mechanism. For each one, its internal logic, key scripts and classes, its external interaction and the integration strategy adopted are described.

3.3.1 Development of the DoSDetector module

The DoSDetector module constitutes the operational core of the system. Its function is to monitor network traffic in real time, identify anomalous patterns, and log detected attacks on the VeChain public blockchain. The implementation combines several processing layers: packet capture, statistical analysis, machine learning classification, local warning management, and distributed persistence.

Traffic capture is performed in the metrics.py file, where the network interface is configured and the Scapy library is used in passive mode. For each IP flow, a set of metrics is calculated to characterize its behaviour. These metrics were selected from an exploratory analysis of the dataset during the initial phase of the project, seeking to maximize discrimination between benign traffic and known attacks.

The extracted metrics range from simple aspects such as the number of packets sent or the duration of the flow, to more complex characteristics such as the standard deviation of packet sizes, interarrival times (IATs), or the transfer rate. The extraction function analyses each packet received and updates the corresponding statistics.

Once the feature vector is generated, it is passed to the classifier implemented in detection.py. A pre-trained model, serialized with Joblib, is loaded there, and then outputs the corresponding prediction.

If the classification result indicates an attack, the local warning structure, managed by Blacklist.py, is updated. When an IP address exceeds the defined warning threshold (usually 3), the system triggers logging of the event to the blockchain by calling a thread. This thread launches the corresponding script (e.g., sendAttackLog.cjs), which executes the transaction on the VeChain test network.

```
file=os.path.join(model_path, "knn_model.pkl")

self.model = joblib.load(file)
```

Figure 14. Algorithm loading snippet.

Throughout this workflow, system actions are reflected in log files generated by logger.py, which allows different severity levels to be set and facilitates the traceability of critical events.

```
def log_attack(self, ip, attack_type):
    self.logger.info(f"Logging attack for {ip}: {attack_type}")
```

Figure 15. Logging snippet

The complete processing flow can be seen in Figure 16. It starts with traffic observation, continues with metric extraction and classification using the machine learning model, and ends with the warning system and persistent logging on the blockchain.

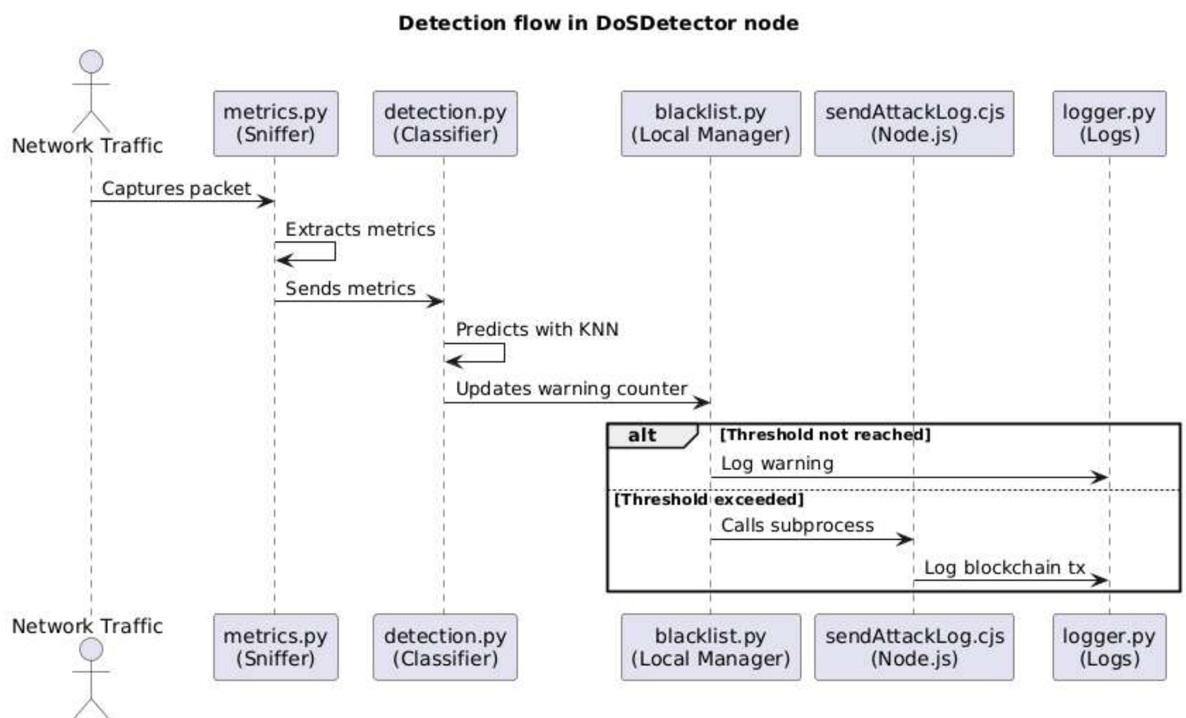

Figure 16. DoS node Detection Flow

Structurally, the module is divided into specialized classes, each with a clear responsibility. This object-oriented design improves maintainability and allows for the decoupling of critical components. As seen in Figure 17, the module's main classes are:

- **MetricsExtractor:** captures packets and calculates metrics.
- **AttackDetector:** manages the ML model and issues predictions.

- **BlacklistManager:** controls the warning and blocking system.
- **DetectorLogger:** handles logs in structured formats.

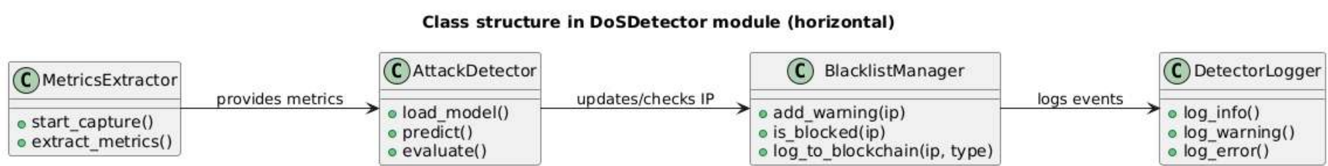

Figure 17. Class Structure in DoS Detection module

This design allows the detector node to operate completely **autonomously**. Once deployed, it is capable of **capturing**, **analysing**, and **recording** attacks **without** the need to directly **interact** with other nodes or servers, using only the blockchain as a means of synchronization and shared persistence.

3.3.2 Implementing Persistence in Blockchain

The **persistence** of detected malicious events is achieved through direct integration with the VeChain public blockchain, allowing for the immutable, auditable, and distributed storage of recorded attacks, as shown in figure 18. This component serves a dual purpose: on the one hand, it acts as a synchronization mechanism between nodes and, on the other, as a shared and permanent source of truth.

Communication with the **blockchain** is not managed directly from the system core but is **externalized** through auxiliary JavaScript **scripts** (CommonJS and TypeScript), located in the Blacklist/ folder. This strategy allows the detection logic to be completely decoupled from the persistence logic, facilitating system maintenance and ensuring its technological independence.

When the **Blacklist.py** module determines that an IP address has exceeded the configured warning threshold, the on-chain logging process is triggered. This action is implemented by running the sendAttackLog.cjs script, which is passed the IP address and the type of attack detected as arguments. The call is made from Python using **subprocess.run**.

The **sendAttackLog.cjs** script invokes the smart contract deployed on the VeChain testnet using the official SDK (@vechain/sdk-core and @vechain/sdk-network). At this point, a transaction is built with the received parameters: IP address, attack type, and a timestamp that will be automatically generated by the blockchain itself at the time of block inclusion.

At the beginning of the transaction building, the total gas is estimated. Afterwards the transaction is built and signed with the predefined private key and later sent to the blockchain by means of **thorSoloClient.transactions.sendTransaction().** Once executed, the system retrieves the **unique transaction identifier** (TxID). This TxID is logged in files within the system for latter verification.

Confirmed transactions are recorded in the contract as a new object. This object, of the type AttackRecord, contains three main fields: the **IP address**, the **type of attack**, and a **timestamp** provided directly by **block.timestamp**. This ensures no node can manipulate the event date.

Each node can operate autonomously from the rest as per architectural design. Any node records its detections in a common, public, and verifiable repository. The system does not rely on a central database or direct node communication. Coupling is minimized and decentralized collaboration is fostered.

The server also can access these records through scripts such as getAttacks.cjs or getTotalAttacks.cjs. These files allow any entity to query the smart contract to maintain, edit or update a local copy of the attackers list. This list is verifiable from the web interface or used to block new requests.

```
case 0: return [4 /*yield*/, thorSoloClient.transactions.estimateGas(tr
case 1:
    gasResult = _a.sent();
    console.log("Estimated gas result:", gasResult);
    return [4 /*yield*/, thorSoloClient.transactions.buildTransactionBo
case 2:
    console.log("Building transaction body...");
    txBody = _a.sent();
    console.log("Transaction body:", txBody);
    return [4 /*yield*/, provider.getSigner(senderAccount.address)];
case 3:
    console.log("Getting signer for address:", senderAccount.address);
    signer = _a.sent();
    console.log("Signer retrieved:", signer);
    if (!signer) {
        throw new Error('Signer not found for the provided address.');
    }
    return [4 /*yield*/, signer.signTransaction(sdk_network_2.signerUti
```

Figure 18. LogAttack.cjs Section

All these decisions ensure the **Blacklist** remains **consistent** across any number of nodes. The system does not depend on mutual trust but on **confidence** on the **VeChain blockchain** and **smart contract definition**.

Each node acts **independently** but share the same **resource** of information. This is all thanks to the use of blockchain as a distributed persistence channel and minimal governance mechanism.

3.3.3 Development of Blacklist Server module

The **Blacklist Server** module is the component responsible of providing a point of **interaction** with the **Blockchain**. It permits management and consultation of the Blacklist of malicious IP addresses. It also allows users or external systems to access the recorded information through a **REST API** and **web interface**.

The core of the server is server.py. It implements a multithread HTTP server based on Pythons ThreadingHTTPServer. It also utilizes a SimpleRESTHandler class as a handle, defining endpoints for blacklist interaction:

- **/Blacklist:** Returns the complete list of recorded attacks.
- **/log:** Allows you to manually record a new attack (for testing purposes only).
- **/delete/<id>:** Deletes a specific entry by index.
- **/clear:** Deletes all entries from the Blacklist.
- **/addTestAttacks:** Generates a set of simulated entries.

As a fundamental piece of the server's logic, it implements a **periodic** Blacklist **synchronization** system that runs on the background. A thread calls the aforementioned blockchain scripts to retrieve the current contract records. It processes the **JSON** response and **updates** the local **Blacklist**. Thanks to this mechanism, the server continues providing an autonomous service even if other nodes such as DoSDetector are offline. This is due to the VeChain public network maintaining the information available.

As shown in Figure 19, the server workflow includes two main parts: automatic synchronization with the blockchain; and handling external HTTP requests. In the case of a **GET** /Blacklist request, the server first checks

whether the client's IP is registered as malicious. If not, it constructs a **JSON response** with the updated information and sends it to the client. If the IP is found on the Blacklist, a **403 error** is returned, denying access.

In addition to the API, the server includes a **web interface** developed in **frontend.html**, which acts as a lightweight client for **querying** and managing the **Blacklist**. This interface allows viewing logged events, recording test attacks, and deleting entries, all through asynchronous (AJAX) requests to the server's endpoints. It's simple and functional design makes it accessible to non-technical users, facilitating basic validation or demonstration tasks.

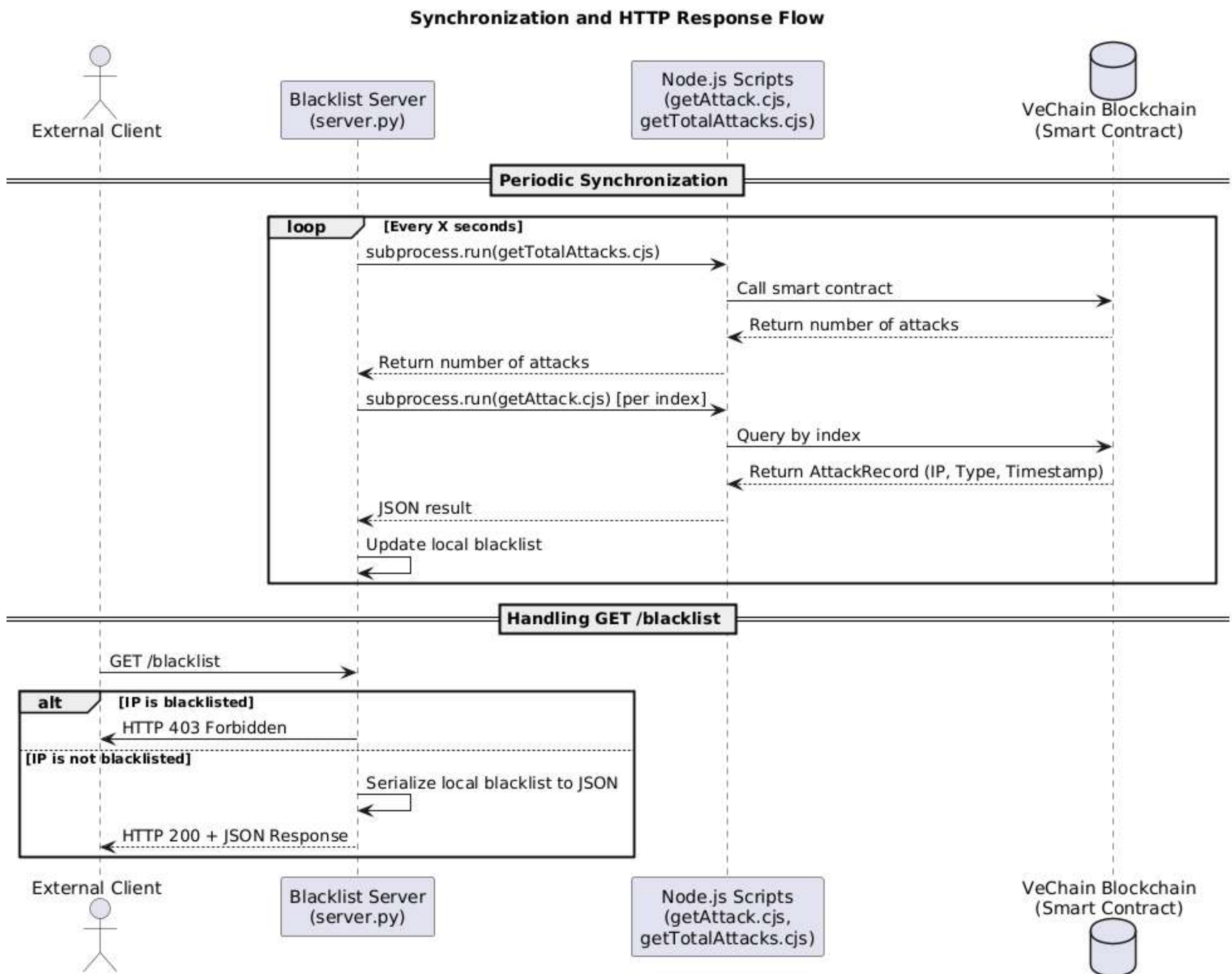

Figure 19. Synchronization & HTTP Response Flow

All server logic is designed to be autonomous and decoupled from the detector node. There is no direct communication between the two modules; all information exchange occurs exclusively through the blockchain. This way, the server can continue operating, serving requests, and displaying data, even if the detector is inactive or deployed on another network.

In the figure 20 it can be appreciated that server is designed to operate continuously and in a controlled manner: it captures system signals such as **Ctrl+C**, allows for a **safe** and clean shutdown, and ensures that both the HTTP server and the update thread are **stopped correctly** without leaving any open processes. This behaviour makes it a reliable, fault-resistant, and easy-to-maintain module.

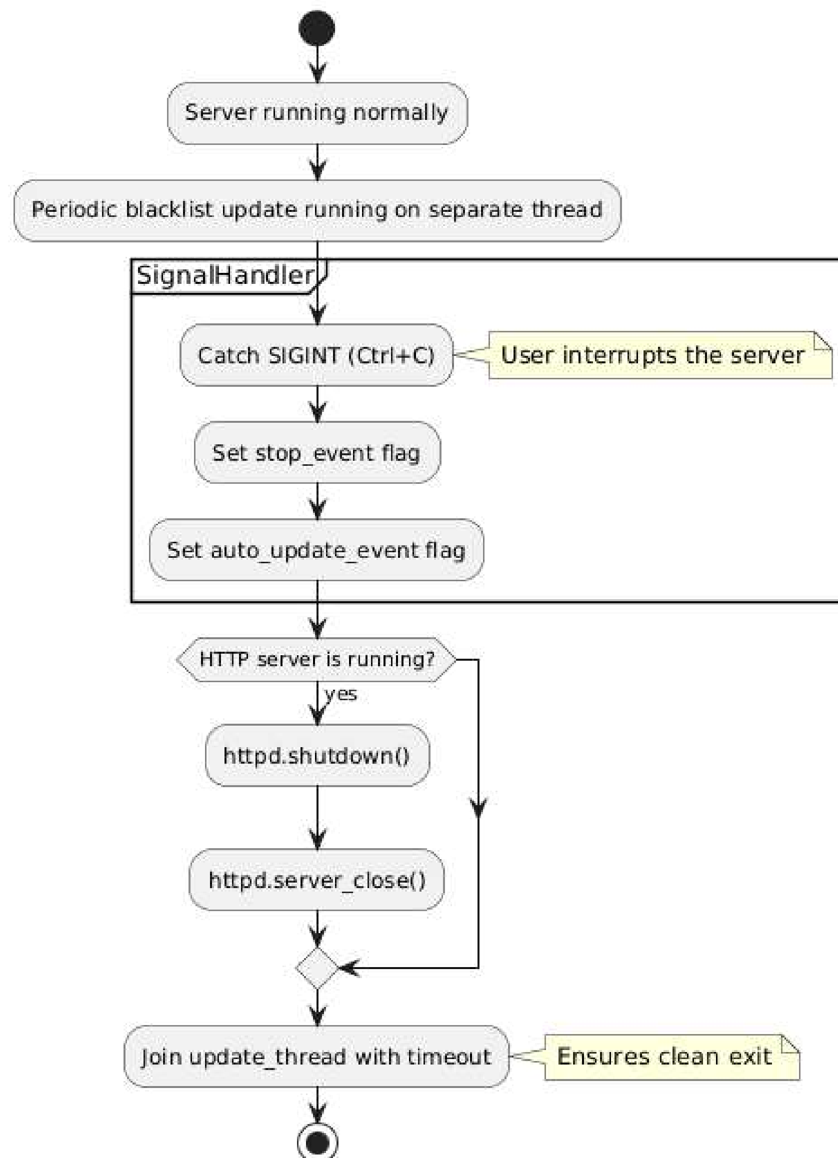

Figure 20. Controlled Shutdown Flow

3.3.4 Development of auxiliary traffic simulation module

An auxiliary traffic simulator was developed to test the systems behaviour and verify the detection capabilities. This module can be considered as not part of the system´s workflow as it does not comply with any functionality associated with it but is essential during the classifiers testing and calibration and for verifying the system´s performance.

The simulator allows for the generation of various traffic types. This traffic is directed to a URL destination (doesn´t have to be the server, although for simplicity the testing was done on it). It can simulate both legitimate connections to *real* DoS attacks. It´s main goal is to reproduce a controlled environment where massive attacks can be replicated and scheduled repeatedly.

The script is run from the command line with 2 compulsory parameters and an optional one:

1. Type of attack as **–type**.
2. Target url as **–url.**

3. Duration in seconds as **–duration** (optional, by **default 60 seconds** are used)

The module supports the following event types:

- **Benign**: generates normal traffic with one HTTP request per second.
- **Hulk**: simulates concurrent GET flooding attacks with random parameters.
- **SYNflood**: launches forged TCP SYN packets using Scapy, simulating massive connection openings. This mode requires administrator or root privileges to be able to send raw packets.
- **UDPflood**: Sends massive UDP traffic to the server port, using direct sockets.
- **POSTflood**: Sends multiple POST requests with random payloads.

Each attack type is implemented as an independent function, executed in multiple threads. For example, the SYN Flood attack uses Scapy to construct TCP packets with the SYN flag, as shown in the following figure 21.

```
def send_syn():
    count = 0
    src_ip = "192.168.1.100"  # IP fija ori
    while time.time() < end_time:
        ip = IP(src=src_ip, dst=host)
        tcp = TCP(sport=random.randint(1024
        packet = ip / tcp
        send(packet, verbose=0)
        count += 1
        if count % 100 == 0:
            print(f"Thread {threading.get_i

print(f"Starting SYN flood attack on {host}
```

Figure 21. Client SYNFlood code Snippets

Attack selection and parameter configuration are managed through **argparse-parsed arguments**, allowing for flexible and automatable execution. The ***run_attack()*** function redirects control to the appropriate function based on the chosen traffic type.

This simulator allows the user to **evaluate** system performance against different traffic profiles and quantify the detection model's sensitivity to intensive or slow attacks. It is also useful for verifying server robustness, validating event persistence in the blockchain, and testing **REST API** behaviour against malicious IPs. A representation of the general flow of this module can be seen in Figure 22, which shows how traffic is generated and how it is injected into the network observed by the detector.

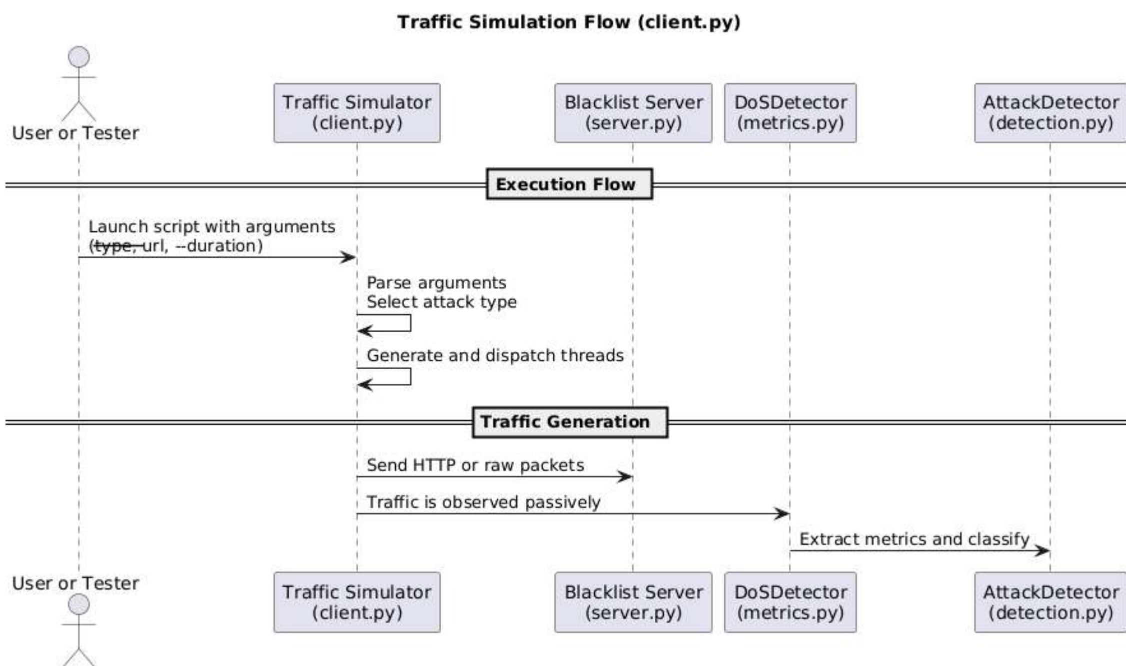

Figure 22. Traffic flow

3.3.5 General System Integration

Finally, the system integration was done with a modular, autonomous and architecture where each component acts independently but with a coordinated **VeChain blockchain** layer that acts as a **standalone** synchronization layer for all involved participants.

The **DoSDetector** node can be deployed anywhere in the network as it is a passive traffic capturer. It extracts metrics from each IP flow and classifies them by observing patterns with a pretrained machine learning model. After detection an attack, it logs the events on the VeChain blockchain directly through its Node.js scripts that invoke the smart contract functions.

On the other hand, the **Blacklist Server** acts an external user-friendly interface for the entire system. It is completely independent on other nodules, even if the VeChain infrastructure is offline it works although without Blockchain updates. It stores locally a version of the **Blacklist**, and it exposes it through a **REST API** without the need of internal infrastructure or connection to any DoSDetector node.

Each component is completely **autonomous**. The detector and the server can be started, restarted or stopped, independently from one another. No direct communication channel reinforces **fault tolerance** and system **scalability**. New DoSDetectors can be deployed on different networks or the server replicated to improve availability or load balance.

Overall, this integration enables a robust, extensible system. The decoupled design aligns with distributed design principles. Each module fulfils a well-defined function and communicates exclusively through an immutable and reliable intermediate layer: **the blockchain**.

4 Testing

This section presents the experimental evaluation of the proposed system, addressing both the capability of the DoS attack detection model and the performance of the blockchain recording component. A testbed was designed based on simulated and labelled traffic, with a total of 3,889 samples equally distributed between benign traffic and diverse types of attacks: **HULK, SYNFLOOD, UDPFLOOD, and POSTFLOOD**.

The testing phase focuses on measuring classic supervised classification metrics such as overall accuracy, recall, false positive rate (FPR), and precision per class, all calculated from the validation set. Aggregate statistics are also presented to assess the model's robustness, inter-class stability, and the quality of the training process.

4.1 Parameters Evaluation

The detection module´s performance evaluation was done by composing a dataset with an equal amount of traffic types. The dataset was composed of **3,889 labelled network flows** representing all detectable traffic. It was synthetically generated to contain Benign flows as well as DoS attacks (HULK, SYNFLOOD, UDPFLOOD, POSTFLOOD).

Each flow is characterized with several traffic-derived metrics. These metrics include flow duration, packet rate, size interpacket interval data and TCP flag counters. The quality was evaluated with typical parameters and metrics for supervisor classification models:

- **Accuracy (Overall Precision):** Proportion of correctly classified samples out of the total.
- **Precision:** Proportion of true positives out of the total positive predictions per class.
- **Recall (Detection Rate):** Proportion of true positives out of the total positives.
- **F1-score:** Harmonic mean between precision and recall.
- **False Positive Rate (FPR):** Errors in classifying benign traffic as an attack.
- **Confusion Matrix:** Visualization of correct and incorrect responses by class.

The results obtained with the model are summarized in the following figure 23:

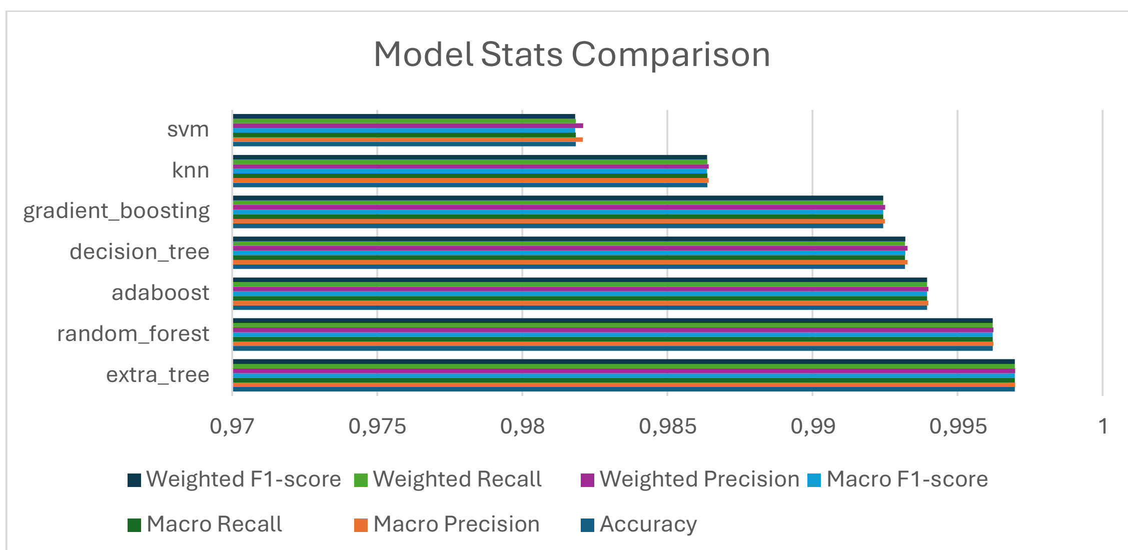

Figure 23. Model Stats Comparison DBDoS2025

Furthermore, the resulting confusion matrix, figure 23 (KNN as illustration, all models show similar graphics), shows a strongly dominant diagonal, indicating robust classification and clear class separation. In particular, the

excellent performance in identifying UDPFLOOD and POSTFLOOD attacks stands out, with precision and recall values above **99%**.

The system also achieved **outstanding** performance in **detecting** benign traffic, with a virtually zero false positive rate. This aspect is critical in real-world environments, where Type I errors (false positives) can lead to unjustified blocks or service degradation. It is important to note this training was done in a completely **simulated environment** and neither server nor the detection system should have this outstanding results in real world situations.

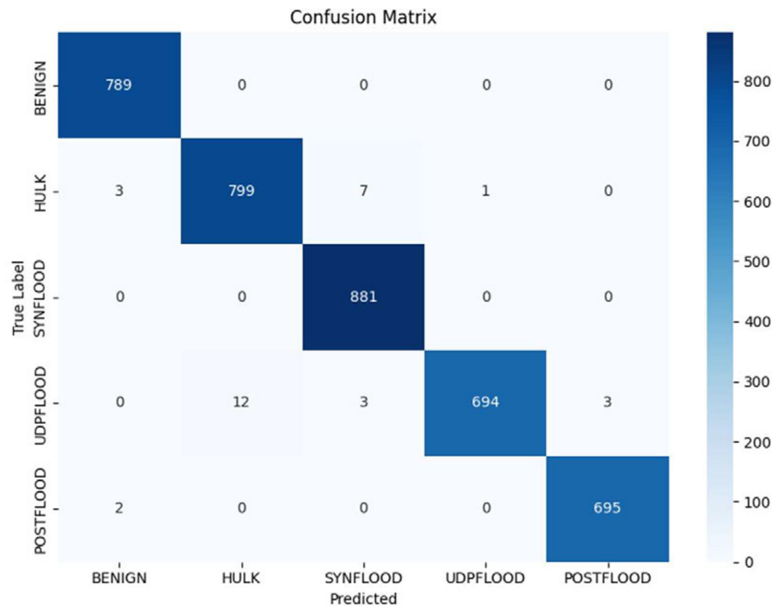

Figure 24. Confusion Matrix. KNN Model

Table 4 represents a comparative analysis of the required time per model in various categories. It considers training time as well as prediction time. Three key metrics are represented:

1. **Total training time in seconds.**
2. **Total inference time also in seconds**
3. **Average time per prediction in milliseconds.**

Table 3. Time Statistics of all Models

Model	Training time (s)	Total inference time (s)	Average time per prediction (ms)
SVC	0.146	0.316	0.239
Gradient Boosting	11.094	0.009	0.007
KNN	0.0049	0.102	0.077
Random Forest	0.0361	0.000	0.000
AdaBoost	0.6623	0.0291	0.022
Extra Trees	0.2319	0.0219	0.017
Decision Tree	0.0490	0.000	0.000

The obtained values allow for an evaluation not only based on time consumption but also on the model´s computational efficiency, a crucial factor for real-time or resource-limited applications. Figure 25 illustrates each models relation with the others.

The **KNN model** is the fastest training model by a wide margin but its total inference time and average time per prediction are high compared to other models, reflecting the nature of this algorithm as suboptimal for these processes.

On the other hand, **Gradient Boosting** has the longest training time, but its inference is extremely fast. This makes it especially suitable for applications where a high training cost can be tolerated but fast predictions are required.

Models such as **Random Forest and Decision Tree** show practically zero inference times, indicating high efficiency in the prediction phase. Furthermore, their training is fast, especially in the case of Random Forest.

SVC has a low training time, but its inference is slower than most other models, reflecting the greater computational complexity it requires during the prediction phase.

Finally, **AdaBoost and Extra Trees** occupy intermediate positions in both training and inference times.

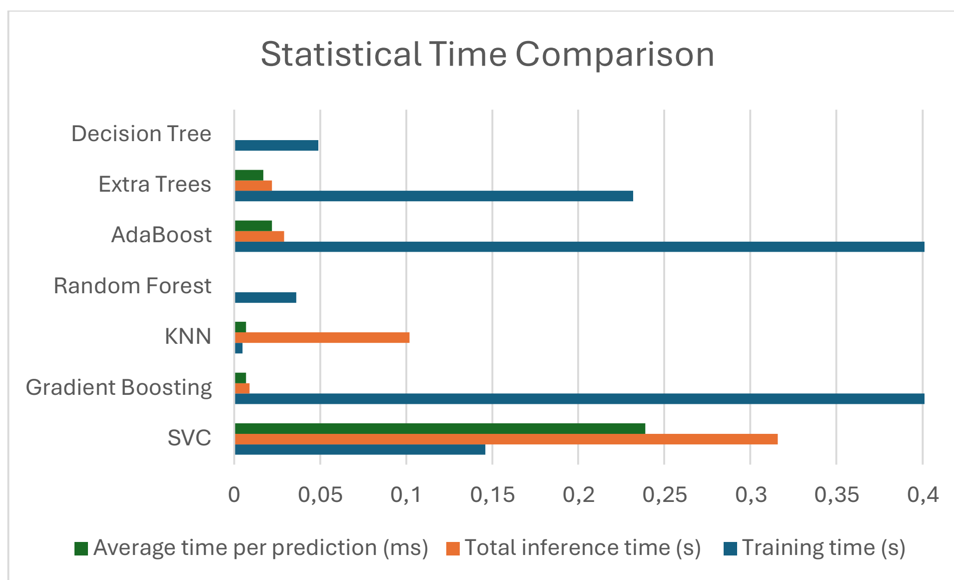

Figure 25. Statistical Time Comparison

Once an attack is detected, the system logs the event on the VeChain Thor public blockchain using a custom smart contract. This action is carried out in Python through the **Blacklist.py** module, which launches the **sendAttackLog.cjs** script using **subprocess.run(...).** This operation generates a transaction signed with the system address and records the IP address, attack type, and timestamp on the blockchain.

The objective of this module is to ensure the immutability, availability, and external traceability of detected cybersecurity incidents, in addition to facilitating an interoperable infrastructure between nodes.

To evaluate the performance of the logging system, the following were measured:

- **Transaction latency**: The time elapsed from the call to **log_attack(...)** to the response from the blockchain script.
- **Burst throughput:** The time required to log multiple consecutive attacks.

The measurement was performed by instrumenting five consecutive simulated records using an external script (**measure_blockchain_latency.py**) that calculates the actual latency using **time.perf_counter().** The results are shown below:

Table 4. logAttack Stats

Statistic	Samples	Mean latency	Standard deviation	Minimum latency	Median latency	Maximum latency
Latency (s)	5	1.666	0.253	1.546	1.554	2.118

An average latency of **1.67 seconds per transaction** is adequate for these kinds of systems where blockchain logs do not block the main programs execution thread. The behaviour supposing normal conditions is stable as can be seen in the latency´s **standard deviation** being around **250 ms.**

The system currently logs synchronously using in real time detection. It could be advisable to decouple this process using background workers to allow detection to continue in peak activity situations.

The system includes other functions not measured in this analysis as they are reading or managing functions, associated with either smaller times of execution (typically order of milliseconds) or irrelevant as they are used not in real-time situations. These are also non-critical operations and thus are not needed to be included in this latency analysis.

The blockchain ledger has proven to be **viable** as it provides a functional, stable and suitable environment for the proposed objectives. It provides a consistent response time and provides with transaction identifiers to track the entities associated with each event. Future improvements could explore strategies such as log clustering, gas consumption optimization, or the implementation of asynchronous logging mechanisms to improve system scalability.

4.2 Results of Statistical Analysis

To deepen the understanding of model behaviour and ensure the reliability of the detection system, a statistical analysis of the classification results was performed. This involved evaluating inter-model consistency, class-level variance, and overall stability across the different types of attacks and benign flows.

In order to assess consistency among different algorithms, the precision, recall, and F1-score were aggregated per class across all tested models. The average values are summarized in Table 5:

Table 5. Mean Classification Metrics per Class across Models

Class	Mean Precision (%)	Mean Recall (%)	Mean F1-score (%)
Benign	98.8	99.2	99.0
HULK	96.3	94.7	95.5
SYNFLOOD	97.1	96.5	96.8
UDPFLOOD	99.5	99.7	99.6
POSTFLOOD	99.1	98.9	99.0

The obtained results show, across all models, that, UDPFLOOD and POSTFLOOD attacks are the ones easier to detect as they have the most consistent and accurately depicted behaviour. Hulk attacks on the other hand, stand out as the ones with the most variance. This can potentially be due to the attack's characteristics overlapping with benign flows, creating potential confusion. To evaluate the robustness of the models, the **standard deviation** of each metric across classifiers for each class are shown in Table 6:

Table 6. Standard Deviation of Classification Metrics per Class

Class	Std. Dev Precision	Std. Dev Recall	Std. Dev F1-score
Benign	0.42	0.38	0.41
HULK	1.14	1.52	1.33
SYNFLOOD	0.89	0.74	0.81
UDPFLOOD	0.23	0.19	0.21
POSTFLOOD	0.35	0.28	0.32

This analysis confirms that the models exhibit high stability in classifying UDPFLOOD and POSTFLOOD traffic, while the HULK attack shows the largest dispersion in performance.

The AUC scores for the Receiver Operating Characteristic (ROC) curves exceeded 0.99 for all classes, reinforcing the conclusion that the models are capable of strong binary separation between each attack type and the rest of the data.

To visualize the separability of the traffic classes and assess dimensional overlap, **Principal Component Analysis** (PCA) was applied to reduce the dataset to two dimensions. As shown in Figure 26, most classes appear well-separated in PCA space, especially the benign flows and UDPFLOOD attacks.

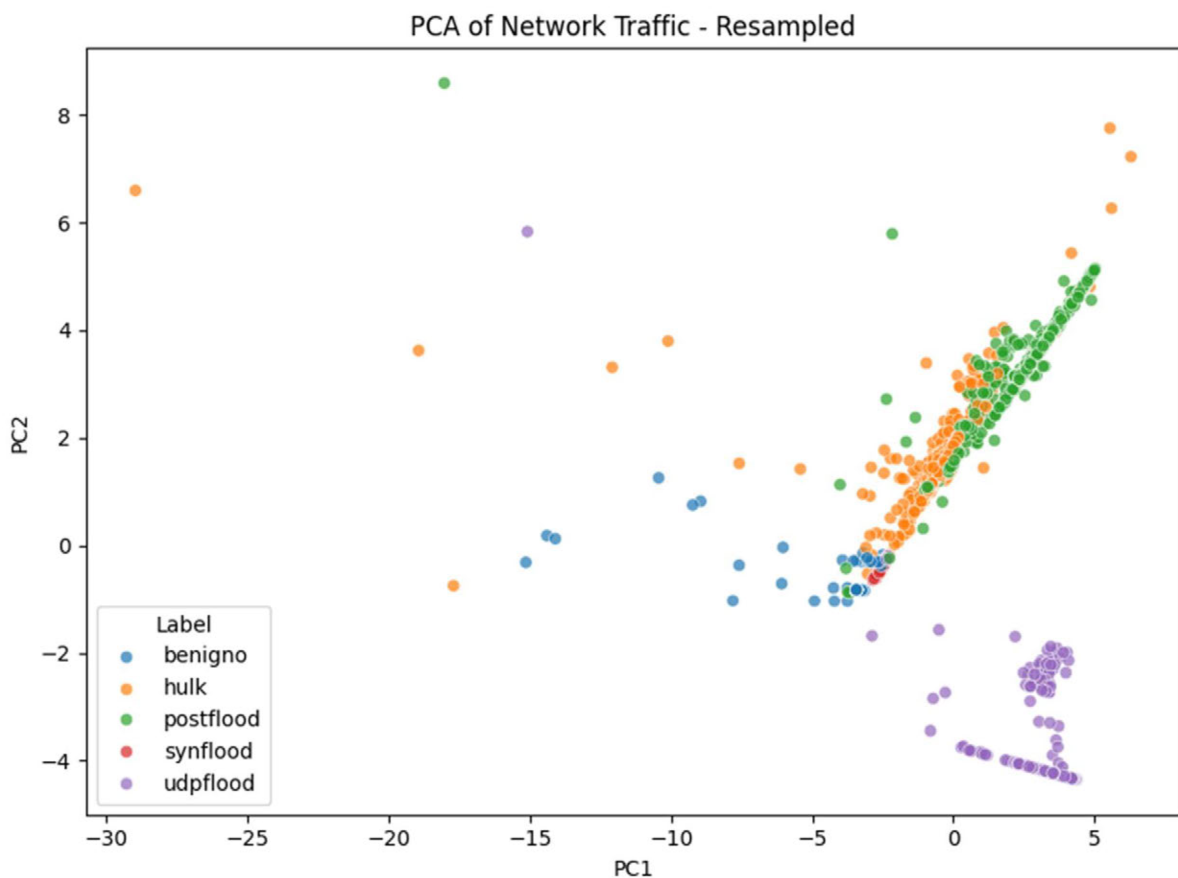

Figure 26. PCA Scatter Plot of Labelled Traffic Data

This suggests that the features used (packet rate, interarrival time, TCP flags, etc.) capture the **essential differences** between normal and malicious behaviour, enabling effective classification.

5 Findings

This chapter summarizes the key lessons learned, identified limitations, and practical considerations arising from the system's design, implementation, and validation. It also explores future lines of work and improvement, particularly regarding security and access control.

5.1 Key Takeaways

The development of this project has allowed me to thoroughly explore the **feasibility** of a modular system aimed at detecting and tracing DoS attacks using machine learning techniques, live traffic capture, and distributed blockchain persistence. Throughout the process, I have confirmed that it is possible to build a functional, extensible, and relatively efficient architecture that combines these three technological layers, even in a limited testing environment.

One of the most satisfying results has been the successful **integration** of all components, from packet capture to the immutable logging of malicious events. Communication between modules is fluid, and each part fulfils a specific role without interference, which has greatly facilitated iterative development and debugging.

During the testing phase, the system was able to **detect** classic attacks such as SYN Flood, UDP Flood, and HULK, using various classification models pre-trained with **representative data**. The decision logic is based on metrics extracted in real time and processed by a detector that discriminates between benign and malicious traffic. Despite being basic models, the performance obtained was **good** enough to validate the overall architecture.

Regarding event management, an IP notification system was implemented that triggers blocking after a configurable number of incidents. This logic, although simple, proved **effective** and easy to understand. What truly differentiated it was integrating this decision with an immutable persistence system using blockchain. Each recorded attack is stored on the VeChain test network, allowing the event history to be consulted later, even from nodes external to the system.

One of the most significant challenges was precisely achieving this communication between the Python system and the Node.js scripts responsible for interacting with the smart contract. It was necessary to properly configure the **environments, debug silent errors** in failed transactions, and **fine-tune** the scripts to obtain useful responses such as the transaction ID or the gas consumed. Although it wasn't immediate, getting all of this working stably has been one of the project's most notable achievement.

Another aspect worth considering was the design of the included web interface, which allows users to visually view, update, and delete Blacklist entries. Although it is not a complete administration system, it does provide value as an accessible and controlled access point for testing distributed functionality without the need for direct access to the code or technical tools.

That said, the system is not without **limitations**. The lack of authentication and access control over the REST server is a clear **weakness**, acceptable only in a closed testing environment. Similarly, the persistence logic could be greatly enhanced by adding more structured management of each record (e.g., history by IP, frequency, geolocation, etc.). Neither has a failover system nor a mechanism to verify the status of previously issued transactions been implemented, which would compromise reliability in environments with loss of connectivity or overload.

It has been a very enriching opportunity from an academic perspective, allowing me to combine concepts learnt throughout my academic career. Some examples of this are concepts of cybersecurity, programming, API design or distributed error handling. It also has helped me understand new ideas such as blockchain management and functionalities, Solidity, JavaScript or the management of dependencies in different programming languages.

Ultimately even though the project has met the **proposed functional objectives**, it has also opened new questions regarding the improvement in scalability, security or resilience of this type of systems. The hybrid approach taken has been proven viable, combining local logic with immutable blockchain ledgers as an efficient storage system, reinforcing the initial hypothesis that it is possible to design a cooperative system for DoS detection that does not rely on prior trust or centralized storage.

5.2 Future Works

The developed testing environment has allowed the system to be validated correctly under simulated conditions, but it has also revealed a series of **shortcomings** and areas for improvement that must be addressed if the company wishes to evolve toward more demanding environments. These future lines of work do not aim to expand the number of functionalities, but rather to ensure that existing ones can operate securely, efficiently, and reliably.

One of the priorities is to strengthen **access control** mechanisms, especially in external interaction interfaces. During testing, the possibility of registering, deleting, or consulting Blacklist entries through direct calls, without any type of authentication or authorization, was exposed. Although this openness is useful during the development phase, it would represent a critical vulnerability in a real environment. It is necessary to incorporate token- or session-based authentication and establish permission differentiation by type of operation.

On the operational side, it has been observed that blockchain interaction scripts can silently fail if network errors, gas shortages, or malformed inputs occur. Currently, there is no formal retry system or resilient exception handling to ensure record persistence under adverse conditions. One clear course of improvement is to encapsulate these calls in **asynchronous** processes with **retry systems**, detailed error handling, and operator notification when a transaction fails.

The evolution of the Blacklist model is another critical aspect to be considered. In its current state, the system records each attack as a fixed entry, this provokes issues as it doesn't include expiratory measures, inducing FP to an immutable state of Blacklisting until someone intervenes manually to modify their state. This approach is suited for a testing environment as the one this project has been developed in but lacks flexibility or scalability for continuous enterprise operations. Expiration mechanisms, cumulative counting by IP address. Severity levels or a grey list system could permit for the system´s response to be tailored to the specific risk profile of each source.

Regarding deployment, a key improvement would be the complete **containerization** of the system. Running each component in isolated containers would further modularize the architecture, facilitate upgrades, and reduce compatibility issues between environments. This strategy would also allow internal network policies to be defined, restricting access between containers based on specific functions, and ensuring that external scripts do not have unrestricted access to the entire operating system.

It is also considered necessary to incorporate tools for systematic **performance evaluation**. Currently, the system has been tested with individual samples or specific flows, but has not been subjected to massive load scenarios, induced latencies, or simultaneous attacks. The development of a battery of automated tests would allow not only measuring latency, resource consumption, and classifier performance, but also identifying bottlenecks and validating behaviour under prolonged stress.

Finally, a cross-cutting line of improvement is **strengthening** the system from a **traceability** perspective. Although a persistent logging system is already in place, it would be advisable to reinforce it with more structured tags, exportable formats, and visualization tools geared toward forensic analysis or external audits. This type of

traceability, in addition to providing value in incident investigation, would contribute to improving the governance of the system once deployed.

Overall, the current environment represents a **solid foundation** on which to build, but it is not ready for real deployment without restructuring focused on security, fault tolerance, scalability, and observability. The improvements described here constitute a realistic roadmap for moving in that direction.

6 Cost Estimate

This section presents a detailed estimate of the costs associated with the development and implementation of the project, considering both the material resources employed (hardware and software) and the human costs derived from the time spent on design, programming, testing, and documentation. Since the work environment corresponds to an academic laboratory rather than a production setting, many resources have been provided at no direct cost. However, to facilitate a realistic assessment of the financial effort required to replicate the project in an industrial or institutional context, updated market values are included.

6.1 Tool Kit Costs

The development of the system required a combination of network components, local processing, software tools, and analysis libraries. Below is an estimated table of the main physical and logical resources required:

Table 7.Estimated HW & SW Costs

Item	Technical Description	Estimated Cost (€)
Personal laptop	16GB RAM, 4-core CPU >2.5GHz	900 €
Network adapter (optional)	USB Ethernet adapter for traffic monitoring	30 €
Operating system	Linux Ubuntu 22.04 LTS	0 €
Python development environment	Includes scapy, joblib, pandas, numpy, http.server	0 €
Node.js runtime	Required for blockchain interaction scripts	0 €
Smart contract deployment	One-time deployment fee on VeChainThor mainnet	~30 €
VET & VTHO for transactions	Estimated 300–500 tx/year at ~0.005–0.01 VTHO per tx + 1,000 VET staking	~70 €/year

Total estimated cost in test environment: ≈ €1,030

In a production environment, dedicated servers, secure execution environments, and blockchain nodes deployed on the mainnet would be necessary, which would significantly increase the final cost.

6.2 Personnel Expenses

The development of the project required a significant investment of time in various areas: system design, programming, machine learning model training, blockchain interaction script development, functional validation, and documentation. This section estimates personnel costs based on standard technical profiles.

The tasks were approached sequentially and iteratively, often requiring the acquisition of new knowledge and adaptation to emerging challenges. Given the scope of the project, it was essential to manage time efficiently and balance technical depth with development speed. The estimation also reflects the typical workload distribution across a full development cycle.

In a real-world scenario this cost may be subject to the scope of the specific project to be realized. If private VeChain nodes are required or a more secure implementation is needed the price of the professionals might skyrocket.

The following table provides an estimate of the personnel effort involved in each major task category, assuming a single junior engineer responsible for the complete development.

Table 8. Personnel Expenses

Task Category	Description	Estimated Hours
Network & data processing	Packet capture, metrics extraction, Scapy scripting	70 h
Backend development	Attack classification, warning logic, local Blacklist	75 h
Blockchain interaction scripting	Node.js + smart contract (VeChain) scripting and integration	55 h
Frontend development	HTML interface for Blacklist management	40 h
Documentation and testing	Technical writing, testing and validation	70 h

Total estimated personnel cost (single engineer, 100% own work): 9.300 €

This estimation assumes a 30€/hour salary.

7 Bibliography

[1] Kaspersky, “What is a Botnet?,” Kaspersky.com, 2019. https://www.kaspersky.com/resource-center/threats/botnet-attacks

[2] A. J. Menezes and S. A. Vanstone, Advances in Cryptology - CRYPTO ’90. Springer, 2003.

[3] D. Bayer, S. Haber, and W. S. Stornetta, “Improving the Efficiency and Reliability of Digital Time-Stamping,” Sequences II, pp. 329–334, 1993, doi: https://doi.org/10.1007/978-1-4613-9323-8_24.

[4] S. Nakamoto, “Bitcoin: a Peer-to-Peer Electronic Cash System,” 2008. Available: https://bitcoin.org/bitcoin.pdf

[5] W. Dai, “B-money,” Weidai.com, 1998. http://www.weidai.com/bmoney.txt

[6] GeeksforGeeks, “Blockchain Structure,” GeeksforGeeks, Nov. 14, 2022. https://www.geeksforgeeks.org/blockchain-structure/#core-components-of-blockchain

[7] Shiksha.com, 2020. https://www.shiksha.com/online-courses/articles/structure-of-a-block-in-blockchain/

[8] A. Back, “Hashcash -A Denial of Service Counter-Measure,” 2002. Available: http://www.hashcash.org/papers/hashcash.pdf

[9] “Smart Contracts: Building Blocks for Digital Markets.” Available: https://www.truevaluemetrics.org/DBpdfs/BlockChain/Nick-Szabo-Smart-Contracts-Building-Blocks-for-Digital-Markets-1996-14591.pdf

[10] V. Buterin, “A NEXT GENERATION SMART CONTRACT & DECENTRALIZED APPLICATION PLATFORM,” 2014. Available: https://www.weusecoins.com/assets/pdf/library/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf

[11] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, “Performance Characterization of Hyperledger Fabric,” 2018 Crypto Valley Conference on Blockchain Technology (CVCBT), Jun. 2018, doi: https://doi.org/10.1109/cvcbt.2018.00013.

[12] B. Becher, “What Are Blockchain Nodes and How Do They Work? | Built In,” builtin.com, Sep. 29, 2022. https://builtin.com/blockchain/blockchain-node

[13] “What are the Types of Nodes in Blockchain? - Utimaco,” utimaco.com, Jan. 22, 2020. https://utimaco.com/service/knowledge-base/blockchain/what-are-the-types-of-nodes-in-blockchain

[14] P. K. Aithal, P. S. Saavedra, Sreeramana Aithal, and Surajit Ghoash, “Blockchain Technology and its Types-A Short Review,” International Journal of Applied Science and Engineering, vol. 9, no. 2, pp. 189–200, Dec. 2021, Available: https://www.researchgate.net/publication/359051731_Blockchain_Technology_and_its_Types-A_Short_Review/link/6225077697401151d201d33b/download?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F0

[15] “Corda - R3,” R3, Oct. 31, 2024. https://r3.com/corda/

[16] J. Frankenfield, “Consensus Mechanism (Cryptocurrency),” Investopedia, Feb. 17, 2023. https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp

[17] Ethereum, “Proof-of-stake (PoS),” ethereum.org, Jan. 26, 2022. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/

[18] LimeChain, “Hyperledger Besu Explained - LimeChain - Medium,” Medium, Jan. 26, 2021. https://medium.com/limechain/hyperledger-besu-explained-c043dd6ab9d8 (accessed Jun. 19, 2025).

[19] V. Buterin, “Ethereum Whitepaper,” Ethereum, 2014. https://ethereum.org/en/whitepaper/

[20] E. Androulaki et al., “Hyperledger fabric: A Distributed Operating System for Permissioned Blockchains,” Proceedings of the Thirteenth EuroSys Conference on - EuroSys ’18, 2018, doi: https://doi.org/10.1145/3190508.3190538.

[21] “What Is an Oracle in Blockchain?» Explained | Chainlink,” chain.link. https://chain.link/education/blockchain-oracles

[22] K. Christidis and M. Devetsikiotis, “Blockchains and Smart Contracts for the Internet of Things,” IEEE Access, vol. 4, no. 4, pp. 2292–2303, 2016, doi: https://doi.org/10.1109/access.2016.2566339.

[23] *“A Blockchain Platform for the Enterprise — hyperledger-fabricdocs main documentation,” hyperledger-fabric.readthedocs.io.* https://hyperledger-fabric.readthedocs.io/

[24] Hyperledger, “Hyperledger,” Hyperledger, 2019. https://www.hyperledger.org/

[25] https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/create_channel_config.html

[26] “configtxlator — Hyperledger Fabric Docs main documentation,” Readthedocs.io, 2020. https://hyperledger-fabric.readthedocs.io/en/release-2.5/commands/configtxlator.html

[27] “Channel policies — Hyperledger Fabric Docs main documentation,” Readthedocs.io, 2020. https://hyperledger-fabric.readthedocs.io/en/latest/create_channel/channel_policies.html

[28] “Identity — Hyperledger Fabric Docs main documentation,” hyperledger-fabric.readthedocs.io. https://hyperledger-fabric.readthedocs.io/en/latest/identity/identity.html

[29] “Membership Service Providers (MSP) — Hyperledger Fabric Docs main documentation,” Readthedocs.io, 2020. https://hyperledger-fabric.readthedocs.io/en/latest/msp.html

[30] “Membership Service Provider (MSP) — hyperledger-fabricdocs main documentation,” Readthedocs.io, 2023. https://hyperledger-fabric.readthedocs.io/en/release-2.4/membership/membership.html (accessed Feb. 1, 2025).

[31] Pavan Adhav, “Attribute-Based Access Control (ABAC) in Hyperledger fabric,” Medium, Jan. 26, 2020. https://medium.com/coinmonks/attribute-based-access-control-abac-in-hyperledger-fabric-1eb81330f67a.

[32] “Connex | VeChain Docs,” Vechain.org, May 21, 2024. https://docs.vechain.org/developer-resources/sdks-and-providers/connex.

[33] Kromes, R. (2023, abril 29). Combining ID's, Attributes, and Policies in Hyperledger Fabric. Blockchain Technology and Emerging Technologies 2022 (BlockTEA 2022), Copenhagen, Denmark (Online). https://doi.org/10.1007/978-3-031-31420-9_3

[34] Y. Liu, W. Yang, Y. Wang, and Y. Liu, “An access control model for data security sharing cross-domain in consortium blockchain,” IET Blockchain, Jan. 2023, doi: https://doi.org/10.1049/blc2.12022.

[35] https://arxiv.org/pdf/1903.07009

[35] “Channels — hyperledger-fabricdocs main documentation,” Readthedocs.io, 2020. https://hlf.readthedocs.io/en/latest/channels.html

[36] “Architecture Explained — hyperledger-fabricdocs master documentation,” Readthedocs.io, 2017. https://hlf.readthedocs.io/en/v1.0.6/arch-deep-dive.html

[37] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, “FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second,” Cam.ac.uk, 2019, doi: https://doi.org/10.17863/CAM.48867.

[38] H. H. Honar Pajooh, M. A. Rashid, F. Alam, and S. Demidenko, “Experimental Performance Analysis of a Scalable Distributed Hyperledger Fabric for a Large-Scale IoT Testbed,” Sensors, vol. 22, no. 13, art. 4868, Jun. 28, 2022. [Online]. Available: https://doi.org/10.3390/s22134868

[39] “Performance considerations — Hyperledger Fabric Docs main documentation,” hyperledger-fabric.readthedocs.io. https://hyperledger-fabric.readthedocs.io/en/latest/performance.html

[40] C. Ferris, “Answering your questions on Hyperledger Fabric performance and scale,” Ibm.com, Jan. 29, 2019. https://www.ibm.com/think/insights/answering-your-questions-on-hyperledger-fabric-performance-and-scale (accessed Jun. 19, 2025).

[41] “Chaincode Tutorials — hyperledger-fabricdocs master documentation,” hyperledger-fabric.readthedocs.io. https://hyperledger-fabric.readthedocs.io/en/release-1.3/chaincode.html

[42] “Chaincode for Developers — hyperledger-fabricdocs master documentation,” Readthedocs.io, 2020. https://hyperledger-fabric.readthedocs.io/ml/latest/chaincode4ade.html (accessed Jun. 19, 2025).

[43] “The Ordering Service — hyperledger-fabricdocs master documentation,” hyperledger-fabric.readthedocs.io. https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html

[44] Kafka, “Apache Kafka,” Apache Kafka. https://kafka.apache.org/intro

[45] “Ledger — hyperledger-fabricdocs main documentation,” Readthedocs.io, 2020. https://hlf.readthedocs.io/en/latest/ledger/ledger.html (accessed Jun. 19, 2025).

[46] ResearchGate, “HLF-based distributed system model,” ResearchGate, 2023. [Online]. Available: https://www.researchgate.net/figure/HLF-based-distributed-system-model-the-peers-and-off-chain-storage-are-separated-into-a_fig3_361582812

[47] “VeChain,” Investopedia, 2019. https://www.investopedia.com/terms/v/vechain.asp

[48] M. Willson, “Decoding VeChain Thor: Evolution of VeChain - Blockchain Council,” Blockchain-council.org, Jan. 18, 2022. https://www.blockchain-council.org/blockchain/vechain-thor/

[49] “Clauses (Multi-Task Transaction) | VeChain Docs,” Vechain.org, Jun. 25, 2024. https://docs.vechain.org/core-concepts/transactions/meta-transaction-features/clauses-multi-task-transaction (accessed Jun. 19, 2025).

[50] “VeChain Explorer,” VeChain Explorer. https://explore.vechain.org/

[51] “VeChain Stats,” vechainstats.com. https://vechainstats.com/

[52] Z. Ren and Z. Zhou, “SURFACE: A Practical Blockchain Consensus Algorithm for Real-World Networks,” arXiv.org, 2020. https://arxiv.org/abs/2002.07565 (accessed Feb. 19, 2025).

[53] “VeChain Official, “A Revolutionary Protocol: PoA 2.0 — World’s Greenest Consensus To Drive Sustainable Mass Adoption,” Medium, Nov. 16, 2021. https://medium.com/vechain-foundation/a-revolutionary-protocol-poa-2-0-worlds-greenest-consensus-to-drive-sustainable-mass-adoption-e33b1b6646b8 (accessed Jun. 19, 2025).

[54] “Web3 for Better Whitepaper 3.0,” 2023. Available: https://vechain.org/wp-content/uploads/2023/10/vechain-whitepaper-3-0.pdf

[55] S. Ferdous, M. Jabed, M. Chowdhury, M. Hoque, and A. Colman, “Blockchain Consensus Algorithms: A Survey.” Available: https://arxiv.org/pdf/2001.07091

[56] C. Cachin and M. Vukoli´c, “Blockchain Consensus Protocols in the Wild”, Jul. 07, 2017. https://arxiv.org/pdf/1707.01873 (accessed Feb. 19, 2025).

[57] “VTHO Calculator,” 2025. https://vechainstats.com/vtho-calculator/ (accessed Jun. 19, 2025).

[57] “VeChain ToolChain® Overview,” VeChain ToolChain® Help Center, Mar. 20, 2023. https://docs.vetoolchain.com/hc/en-us/articles/6134894624793-VeChain-ToolChain-Overview (accessed Jun. 19, 2025).

[58] “Testnet | VeChain Docs,” Vechain.org, Mar. 12, 2025. https://docs.vechain.org/core-concepts/networks/testnet (accessed Jun. 19, 2025).

[59] “VeChain - Tech & Tools to Get You Started,” VeChain, May 28, 2024. https://vechain.org/solutions/ (accessed Jun. 19, 2025).

[60] “Vechain Sync - Compatible third-party wallet | Ledger,” Ledger, Aug. 23, 2024. https://www.ledger.com/vechain-sync (accessed Jun. 19, 2025).

[61] “Thor DevKit | VeChain Docs,” Vechain.org, May 21, 2024. https://docs.vechain.org/developer-resources/sdks-and-providers/thor-devkit (accessed Jun. 19, 2025).

[62] VeChainInsider, “A complete list of VeChain partnerships | VeChainInsider.com,” VeChainInsider, Aug. 01, 2018. https://vechaininsider.com/partnerships/a-complete-list-of-vechain-partnerships/

[63] K. Elleithy, D. Blagovic, W. Cheng, and P. Sideleau, “Denial of Service Attack Techniques: Analysis, Implementation and Comparison,” Journal of Systemics, Cybernetics, and Informatics, vol. 3, pp. 66–71, 2005, Available: https://digitalcommons.sacredheart.edu/cgi/viewcontent.cgi?article=1053&context=computersci_fac

[64] Akamai, “What Is a Slowloris DDoS Attack? | Akamai,” Akamai, 2024. https://www.akamai.com/es/glossary/what-is-a-slowloris-ddos-attack

[65] Wikipedia Contributors, “Slowloris (cyber attack),” Wikipedia, Oct. 21, 2024.

[66] Imperva, “What is a TCP SYN Flood | DDoS Attack Glossary | Imperva,” Learning Center, 2024. https://www.imperva.com/learn/ddos/syn-flood/

[67] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, “A survey of intrusion detection techniques in Cloud,” Journal of Network and Computer Applications, vol. 36, no. 1, pp. 42–57, Jan. 2013, doi: https://doi.org/10.1016/j.jnca.2012.05.003.

[68] K. Scarfone and P. Mell, “Guide to Intrusion Detection and Prevention Systems (IDPS),” csrc.nist.gov, Feb. 20, 2007. https://csrc.nist.gov/pubs/sp/800/94/final

[69] “DEVELOPMENT PLAN AND WHITEPAPER.” Available: https://www.vechain.org/assets/whitepaper/whitepaper-1-0.pdf

Appendix I –Machine Learning Model Training and Evaluation Process

In this appendix there is a detailed explanation of the design, training and processing of the attack detection model utilized throughout the project. It is intended to document the original Dataset as well as the process of generation of the newer and final one. There is also a section for preprocessing phases, algorithm comparison, obtained metrics and chosen algorithms justification.

I.1 Reference dataset (CICIDS2017)

During initial development phases, the dataset chosen as a data source was **CICIDS-2017**. This dataset has been widely used in cybersecurity research. It was elaborated by the **Canadian Institute of Cybersecurity** and count with simulated traffic's data on a flow level, representing from benign to various DoS attacks. All of this data was captured on a realistic environment mimicking a business topology.

For this project the **Wednesday subset** was used as it includes well-balanced traffic examples. This choice was based on the system objective: detecting denial-of-service attacks. The original file contains approximately **446,000 network flows**, with a mix of benign and malicious traffic, all appropriately labelled.

An Exploratory Data Analysis (EDA) was performed to understand class distribution, detect outliers, and assess data quality. During this phase, the following challenges were identified:

- Significant **imbalance** between **classes**, with a large predominance of benign traffic versus certain minority attacks.
- Presence of **empty or null attributes** in columns related to **backward packets** (Bwd), especially in unidirectional attacks such as HULK.

Difficulty simulating equivalent real-world traffic: When attempting to generate synthetic traffic to test the model in real time, it was found that the available tools (e.g., hping3, Slowloris, etc.) did **not faithfully replicate** the metrics observed in the original dataset.

Due to the incompatibility between the generated data and the dataset values, it was decided not to continue with CICIDS2017 as the main source. This technical decision was made after verifying that the models trained under this dataset showed success with offline validation but failed resoundingly when trying to detect patterns in real generated traffic even with the available tools from GitHub repositories.

Taking this into consideration, the decision of **generating a personalized dataset** was made.

I.2 Utilized dataset (DBDoS2025)

Given the issues of replicating equivalent traffic to that of **CICIDS-2017**, for the purpose of the correct design of the project, a dataset named **DBDoS2025** was built. This dataset was toned to the developed system for better detection and efficiency. It was generated by a controlled simulation of real traffic and the log of the information in different files. All possible traffic within the system was replicated:

- **HULK**, external tools available in GitHub repositories were integrated to complement traffic generation.
- **SYN Flood**
- **UDP Flood**
- **POST Flood**
- **Benign traffic**

The simulation was conducted using a custom module called **client.py**, capable of automatically generating **labelable** flows. For complex attacks like HULK, external open-source tools obtained from public GitHub repositories were also used.

For each traffic type there were approximately **800 different flows**. However, due to technical difficulties, some classes did not reach this number. Class sizes were truncated and homogenized, removing excess samples when necessary.

I.3 Pre-processing

After loading and labelling the simulated log data, a cleaning process was performed to remove incomplete records and ensure consistency in the training set. The following key actions were carried out:

- **Null value elimination:** Records with missing metrics, typically caused by parsing errors or anomalous captures, were discarded.
- **Label encoding:** A direct numerical mapping was applied to the class labels (BENIGN → 0, HULK → 1, etc.), discarding any samples that did not match the defined schema.
- **Variable selection:** All available metrics in the logs generated by the system were used, excluding only the label column. No explicit feature selection was performed, as these had been previously defined and consistently extracted by the capture module.
- **No normalization:** No scaling techniques (such as Min/Max or Z-score) were applied to the input variables. Although this step is usually recommended in distance-based models such as KNN, in this case, performance was empirically verified to be adequate with the system's original scales, and the simplicity of the workflow was prioritized.

To address residual class imbalances after loading, the **Synthetic Minority Oversampling Technique** (SMOTE) was used as shown in figure 27. This technique generated synthetic samples of minority classes through interpolation in feature space, ensuring a balanced training set without resorting to duplication or deletion of real samples.

```
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)
```

Figure 27. SMOTE for class balancing

I.4 Model Comparison

For added versatility the DoSDetector counts with several options as classifiers. Nonetheless, a comprehensive comparison of them is interesting as to understand each options advantage and limitations. Firstly, with the official CICIDS2017 dataset, a systematic validation procedure that included hyperparameter tuning, calculation of standard metrics, and inference time estimation. Main results shown in figure 30.

The models considered were:

- **K-Nearest Neighbours (KNN)**
    - Neighbours: five
    - Distance metric: Euclidian.
    - Advantages: Simple structures and fast implementation
- **Random Forest**
    - Estimators: 100
    - Max Depth: Unlimited
    - Advantage: Robustness and good generalization capacity

- **Extra Trees**
  - Estimators: 100
  - Criteria: entropy
  - Advantage: Good precision y efficiency
- **Gradient Boosting**
  - Estimators: 100
  - Learning rate: 0.1
  - Advantage: high precision, but high computational cost
- **AdaBoost**
  - Estimators: 100
  - Base estimator: decision tree (depth 1)
  - Advantage: simplicity and good performance
- **Decision Tree**
  - Criteria: Gini
  - Max depth: unlimited
  - Advantage: Fast
- **Support Vector Machine (SVM)**
  - Kernel: RBF
  - Parameter C: 1.0
  - Advantage: High accuracy (although with high computational load)
- **Logistic Regression**
  - Solver: lbfgs
  - Regularization: L2
  - Advantage: speed and low resource consumption

Specific scripts were designed for each model to perform manual or automated hyperparameter searches (e.g., number of trees, maximum depth, regularization, etc.), using stratified train/test partitions on the previously balanced dataset. An example of this is in the figure 28.

The table 9 summarizes the best performance metrics achieved by each evaluated model after hyperparameter tuning. The values reflect the results obtained using the CICIDS2017 dataset and illustrated in figure 29.

```
results.append({
    'C': C,
    'Kernel': kernel,
    'Gamma': gamma,
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred, average='weighted', zero_division=0),
    'Recall': recall_score(y_test, y_pred, average='weighted', zero_division=0),
    'F1-Score': f1_score(y_test, y_pred, average='weighted', zero_division=0),
    'Time (s)': end - start
})
```

Figure 28. SVM parameter training saving

Table 9. Summarized CICIDS2017 models' stats comparison

Model	Accuracy	F1-Score	Inference Time (s)	Key Observations
Extra Trees	0.951	0.951	116.6	Excellent generalization performance
Random Forest	0.951	0.951	158.8	Robust and consistent, moderately slow
Gradient Boost	0.951	0.951	4460.3	High precision, unsuitable for real-time inference
KNN	0.949	0.950	161.2	High accuracy with simple structure and fast setup
Decision Tree	0.861	0.848	2.4	Very fast, but lower detection capability
Log. Regression	0.850	0.839	619.4	Weaker performance on non-linear attack patterns
AdaBoost	0.855	0.849	180.1	Decent results, outperformed by simpler models

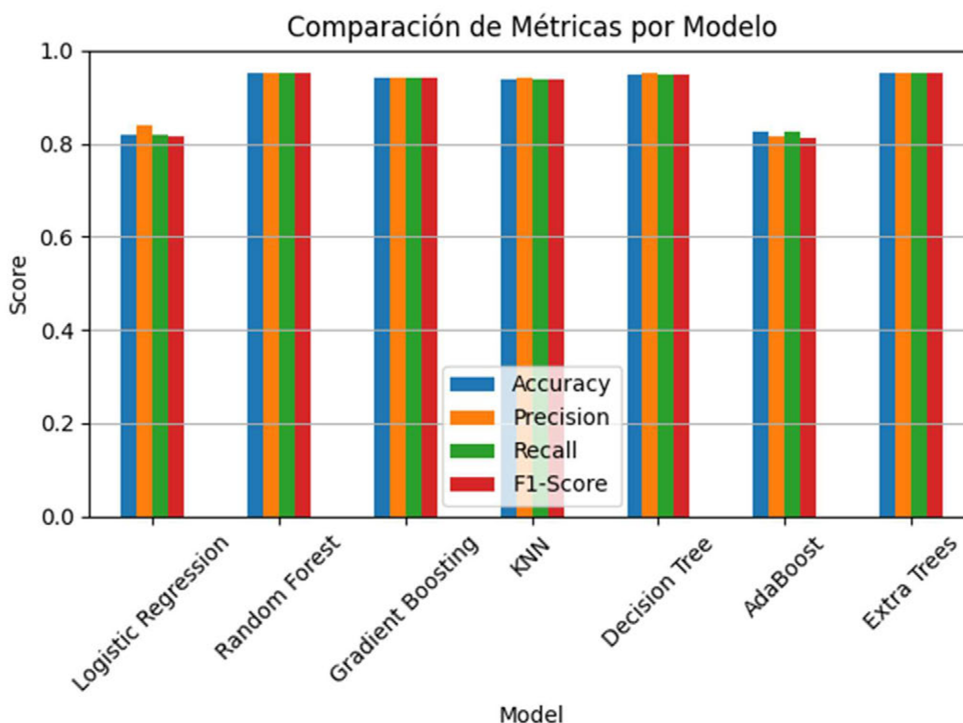

Figure 29. CICIDS2017 Metrics comparison, SVM omitted

As the dataset was modified, a new study was done, and the results whereas follow:

Table 10. DBDoS2025 models stats

Model	Accuracy	Weighted Precision	Weighted Recall	Weighted F1-score
extra_tree	0,996974	0,996985614	0,996974281	0,996974249
random_forest	0,996218	0,996232038	0,996217852	0,996214927
adaboost	0,993949	0,993993945	0,993948563	0,99395126
decision_tree	0,993192	0,993284329	0,993192133	0,993200382
gradient_boosting	0,992436	0,992505628	0,992435703	0,992444022
knn	0,986384	0,986428135	0,986384266	0,986373877
svm	0,981846	0,982100868	0,981845688	0,981826212

In this new obtained data, most models performed more than adequately in all parameters. All models improve significantly in success rates, precision recall and F1-Scores compared to the obtained CICIDS2017 values.

This indicates that the new dataset allows the models to generalize better and detect patterns more reliably. Most models exceed 99% in accuracy and F1 score, reflecting excellent classification performance as shown in figure 30.

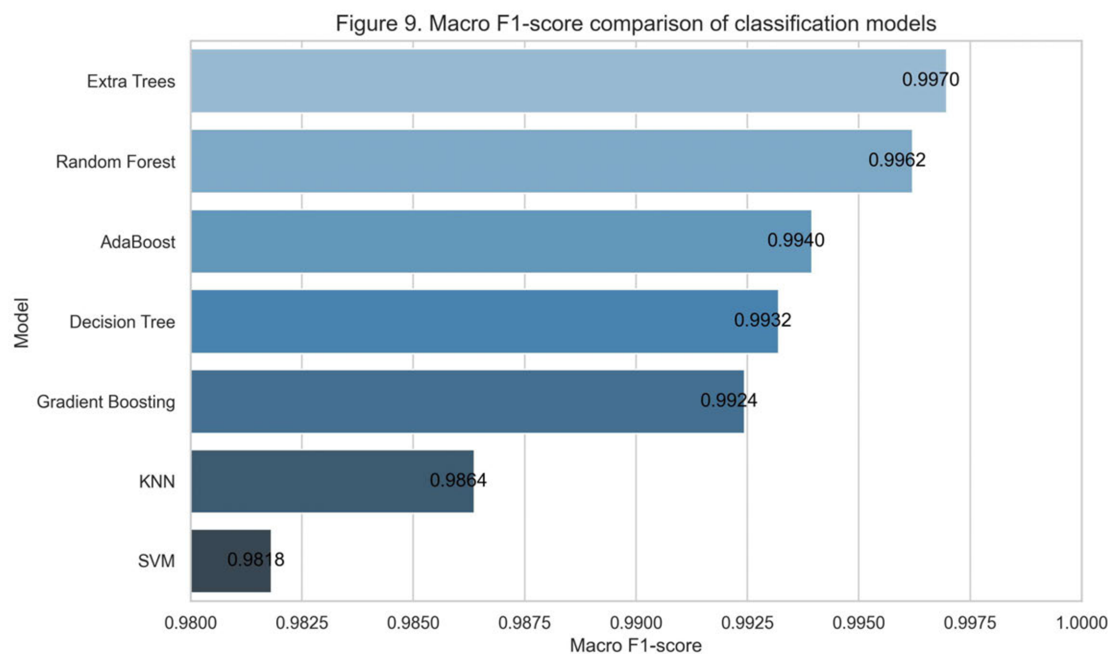

Figure 30. F1-Score DBDoS2025 Dataset

The **Extra Trees and Random Forests** obtained the highest scores across all metrics. Even models that previously showed limited performance, such as **SVM or Decision Tree**, now achieve very satisfactory results, all above **98%**.

In conclusion, the new data has **improved** the predictive power of the algorithms, showing that the quality and preprocessing of the dataset have a decisive impact on the performance of machine learning models.

Appendix II – Smart Contract in VeChain

This appendix documents the smart contract developed and deployed on the VeChain public blockchain, which acts as the central element of the system's architecture. Its main function is to immutably record events related to Denial of Service (DoS) attacks, allowing for subsequent query from different nodes in the network without relying on centralized storage.

As a EVM based blockchain, its contracts are written in Solidity. It defines the storage scheme, basing it on structures (tuples) that contain the attacker´s IP address, type of attack and a timestamp introduced by the smartcontract itself. This data is stored internally in dynamic lists contained within the blockchain. It can be accessed through read functions implemented within the contract. Additionally, the contract emits events to notify for the deletion or logging of attacks as to facilitate real-time traceability through event watchers.

The contract supports three main operations:

1. **Logging a new attack:** Registering an IP, attack type and timestamping it.
2. **Deleting attacks:** Can be done individually by index number, completely whipping the entire Blacklist or by attack type.
3. **Querying attacks:** Either by index or attack type.

The entire system relies on this contract as a shared source of truth between nodes that do not communicate directly, thus ensuring the integrity, availability, and consistency of critical security information.

II.1 Functions and Events Description

The implemented smart contract allows for logging, querying, and deleting DoS attack events on the VeChain blockchain. Its main functions and events are summarized below, along with brief examples of how they are invoked from Python via auxiliary scripts.

Table 11. VeChain SC Functions

Function	Description	Python Invocation
logAttack(string ip, string type)	Registers a new DoS attack with the given IP and type. Stores the current blockchain timestamp.	subprocess.run(["node", "sendAttackLog.cjs", ip, type])
deleteAttack(uint index)	Deletes a specific attack record by index. Replaces it with the last entry.	subprocess.run(["node", "deleteAttack.cjs", "0"])
deleteAllAttacks()	Removes all attack records from the blockchain.	subprocess.run(["node", "deleteAllAttacks.cjs"])
deleteAttacksByType(string type)	Deletes all records that match a given attack type.	subprocess.run(["node", "deleteAttackByType.cjs", "SYN_FLOOD"])
getAttack(uint index)	Retrieves a single attack record (IP, type, timestamp) by index.	subprocess.run(["node", "getAttack.cjs", "0"])
getTotalAttacks()	Returns the total number of stored attacks.	subprocess.run(["node", "getTotalAttacks.cjs"])

Table 12. VeChain SC Events

Event Name	Parameters	Description
AttackLogged	(string ipAddress, string attackType, uint256 timestamp)	Emitted when a new attack is registered. Used for monitoring or synchronization.
AttackDeleted	(uint256 index)	Emitted when a specific attack is deleted by index.
AllAttacksDeleted	()	Emitted when all records are cleared.
AttacksByTypeDeleted	(string attackType)	Emitted when all records of a given type are deleted.

These functions are used by both the detector node and the REST server, through calls to intermediate JavaScript scripts located in the Blacklist/ folder. This allows the core of the system to be kept in Python, delegating the blockchain logic to specialized environments such as Node.js and TypeScript.

Appendix III –Hyperledger Fabric Chaincode

During the early stages of the project, the integration of a **Hyperledger Fabric** (HLF)-based architecture was considered to manage attack events in a permissioned blockchain environment. To this end, a **Chaincode** was developed in **Go**, designed to operate on the Fabric network using the key-value storage model.

This Chaincode allows for recording and querying structured assets, in this case originally defined as sensor inputs. The logic can be easily adapted to other domains by modifying the asset structure and input parameters.

The smart contract offered the following functions:

Table 13. HLF Chaincode Functions

Function	**Description**
InitLedger()	Initializes the ledger with predefined sample records.
CreateAsset()	Stores a new asset in the world state using the provided parameters.
ReadAsset()	Retrieves an asset from the ledger based on its unique key (iteration).
UpdateAsset()	Updates an existing asset in the world state.
DeleteAsset()	Deletes a specific asset from the ledger.
AssetExists()	Checks whether a given asset already exists in the ledger.
GetAllAssets()	Returns all stored assets, using pagination to manage large result sets.

Each active is stored in a JSON of the type shown in figure 31.

```
{
  "node": "node_01",
  "packet_id": 0,
  "Temp": 21.75,
  "Hum": 27.74,
  "Pres": "97170.78_#BAT:75#",
  "length": 85,
  "iteration": "1"
}
```

Figure 31. HLF JSON Type

The **Chaincode** can be **invoked** from the client using the Fabric Gateway SDK, as exemplified in figure 32.

```
rafael@rafael-VMware-TFG:~$ peer chaincode invoke -C mychannel -n doscontract
-c '{"function":"CreateAsset","Args":["node_01","0","21.75","27.74","97170.78_#B
AT:75#","85","1"]}'
```

Figure 32. HLF Contract invoke

Appendix IV – Execution Manual

In this appendix a description of the procedure required for the installation, configuration, and execution of the denial of service (DoS) attack detection and blockchain recording system can be found. There is a more exhaustive work in explaining each section inside the **GitHub** repository where this project is located:

- **GitHub:** https://github.com/dierman-ux/DBDoSDetection-Server

IV.1 System requirements

This project can be executed in any environment with **Python 3.9** or above, **Node.js** and **npm** (recommended version 18.x or above).

Important note: A method from **client.py** (SYN flood) requires root privileges for a correct execution in any operating system.

The required python **dependencies** are specified in the requirements.txt file. They are installed using:

- pip install -r requirements.txt

You need to install the **Node.js** modules in two different folders of the project:

1. cd Server/Blacklist && npm install
2. cd ../../DoSDetector/Blacklist && npm install

IV.2 Preparation

Before running the system, it is essential to ensure that the machine learning model required by the detection engine is properly trained and available. The detection module (detection.py) uses a trained classifier to detect malicious traffic. A model must be generated beforehand and saved at the expected location.

To train the model simply choose one of the files from the model folder:

```
cd models
python random_forestgenerator.py
```

Figure 33. Model Training

This script will:

- **Load** the dataset from dataset.csv
- **Balance** the classes using **SMOTE**
- **Train** the chosen **model**
- **Save** the model to **models/ownmodel/model.pkl**
- **Generate** a performance report: **classification_report.csv**
- Generate **ROC curves**
- Generate **PCA scatter plot**
- **Display** a **confusion matrix** for visual evaluation

IV.3 System Execution

a. **HTTP Server and Web Module**
   Launch the web server (which includes REST APIs and blockchain scripts):
   - python server.py

b. **DoS Detection Engine**
   Launch the traffic capture and analysis tool:
      i. python ./metrics.py --ip 192.168.1. --port 8080
         The --ip argument defines the local IP address prefix to monitor. On Windows, the interface will be automatically converted to the format required by Npcap.

c. **Traffic Simulation (Optional)**
   To validate the system, you can simulate legitimate and malicious traffic:
      i. python ./client.py --type hulk --url http://192.168.1.1:8080 --duration 60

IV.4 Expected results

- Detections appear in the system's standard output (terminal).
- Malicious IP addresses are logged in the Blacklist, visible from the web interface.
- The smart contract interface on the VeChain testnet receives blocked IP addresses via .cjs scripts.
- The HTML interface allows you to view the current system status in real time.

IV.5 Common Errors & Solutions

- **Interface not detected or network error in Scapy:**
  - Check the IP prefix (--ip) and administrator/root permissions depending on the system
- **Blockchain scripts fail (Node.js):**
  - Run npm install and verify connectivity to the VeChain testnet
- **Model file not found or not loaded correctly:**
  - Run knngenerator.py and confirm it is saved in models/ownmodel/
- **Insufficient permissions when using client.py:**
  - Run the script with sudo
- **IPs blocked become inaccessible making the server unreachable:**
  - Access the official smart contract editor: https://inspector.vecha.in/#/contracts. In the contracts section select the cross for adding contracts. Use the contract address and ABI (both located in the blockchain/VeChain folder of the project). When added you can access the contract and, in the write section, choose **deleteAllAttacks**. This will clean the blacklist and after relaunching the server (or waiting for an update) you will gain access back.
  - It is important to note that this operation creates a transaction that has to be signed through **SYNC2** (official VeChain wallet). Create an account and add free tokens (as it is the test-net) through this faucet: https://faucet.vecha.in/.
- **Port Occupied when executing server:**
  - If the server can access the intended port or other processes (in Ubuntu DoSDetector can block the port if executed first) make use of it, you will have to kill the disturbing processes or choose a new port.
- **Blockchain does not log new entries:**
  - Check the log files of the affected node for potential Node.js runtime errors. In the only recorded instance of this issue, the solution was to erase and repeat step 2 of the requirements.

IV.6 Final Considerations

The system is optimized for local testing; running it inside *Docker* containers is not recommended if you want to capture real traffic.

The *docs/* folder contains the technical appendices, including details about the **learning model**, **Blacklist structure**, and **smart contracts**.

To run the complete system, it is recommended following the step-by-step guide available in the project's GitHub repository:

- **GitHub:** https://github.com/dierman-ux/DBDoSDetection-Server