RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITÉ DE M'HAMED BOUGARA, BOUMERDÈS

Faculté des Sciences — Département Informatique



Mémoire de Fin d'Études

Pour l'obtention du diplôme de Licence en Informatique
Spécialité : Systèmes Informatique

A distributed solution for detecting Slow-HTTP-DoS attacks in smart contract blockchain-IoT systems by leveraging Fog Computing

Réalisé par :
Amiri Razine Ilyacine
Abdelatif Abdelmalek ahmed
Amaouch Layachi abdallah

Supervised by : Mohammed Haddadi

Année Académique : 2025 / 2026

Acknowledgements

First and foremost, we thank Almighty Allah for granting us the health, patience, determination and courage necessary to complete this final-year project. The completion of this dissertation represents the culmination of an academic journey during which we have learnt to research, experiment, correct our mistakes and transform a technical idea into a working prototype.

We would like to express our deep gratitude to our supervisor, Mr Mohammed Haddadi, for his support, guidance and advice throughout the course of this project. His feedback helped us to structure our work, clarify the problem and gradually improve the technical and academic quality of the proposed solution.

We would also like to thank all the lecturers in the Computer Science department who contributed to our education during our three-year Bachelor's degree. The knowledge we acquired in programming, systems, networks, databases, IT security and software engineering provided an essential foundation for understanding and developing this project.

We would also like to thank our families for their moral support, patience and constant encouragement. Their trust has been a major source of motivation, particularly during the periods of testing, error correction and thesis writing.

Finally, we would like to thank our colleagues and friends for the exchanges, mutual support and technical discussions that have enriched our thinking. To everyone who has contributed in any way to the completion of this work, we offer our sincere gratitude.

Abstract

The rapid evolution of the Internet of Things has led to the emergence of intelligent environments comprising sensors, gateways, connected devices and distributed services. However, these environments remain vulnerable to a number of security risks, including denial-of-service attacks. Among these attacks, Slow HTTP-DoS attacks pose a particular threat, as they can keep HTTP connections open for a long time by sending incomplete or very slow requests. This behaviour gradually consumes server resources whilst sometimes appearing to be legitimate traffic.

IoT devices generally have limited computing power, memory, energy and bandwidth. For this reason, it is not always possible to run resource-intensive detection mechanisms directly on connected devices. Fog computing provides a suitable solution, as it brings processing closer to the data sources and enables faster detection at the fog nodes. In this project, simulated IoT devices send network flow characteristics to a fog node, which uses a machine learning model to classify traffic into two categories: NORMAL or ATTACK.

When an attack is detected, the event is recorded in a log file in JSON Lines format. A Java programme then calculates the SHA-256 hash of each log line and stores only this hash in a Solidity smart contract deployed on a local Ganache Ethereum blockchain. The blockchain is therefore not used to store the complete network data, but solely to maintain a verifiable proof of integrity. The prototype combines Python, Flask, scikit-learn, pandas, numpy, joblib, Java, Maven, Web3j, Solidity 0.8.19, Ganache and Docker. It demonstrates that an IoT-Fog-Blockchain architecture can contribute to the distributed detection of Slow HTTP-DoS attacks and the preservation of the integrity of attack evidence.

Keywords: IoT, Fog Computing, Slow HTTP-DoS, Machine Learning, Blockchain, Ethereum, Ganache, SHA-256, IT security.

Table of Contents

Acknowledgements **2**

Abstract **3**

List of figures **5**

List of tables **5**

General introduction **7**

1 General context and problem statement **9**

1.1 Introduction 9
1.2 Internet of Things 9
1.3 DoS and Slow HTTP-DoS attacks 11
1.4 Fog Computing 13
1.5 Blockchain and data integrity 13
1.6 Challenges 14
1.7 Project objectives 14
1.8 Conclusion 14

2 Analysis and design of the proposed solution **15**

2.1 Introduction 15
2.2 Needs analysis 15
2.1.1 2.2.1 Stakeholders in the system 15
2.1.2 2.2.2 Functional requirements 16
2.1.3 2.2.3 Non-functional requirements 17
2.3 Overall architecture 17
2.4 Description of components 19
2.1.4 2.4.1 IoT Simulator 19
2.1.5 2.4.2 Fog Node Flask API 19
2.1.6 2.4.3 Machine Learning Model 19
2.1.7 2.4.4 Attack Log JSON 19
2.1.8 2.4.5 Java Blockchain Logger 20
2.1.9 2.4.6 Solidity DataStore Smart Contract 20
2.1.10 2.4.7 Ganache 20
2.1.11 2.4.8 Docker and Docker Compose 20
2.5 Workflow 22
2.6 Technology choices 22
2.7 Status of Docker automation 23
2.8 Conclusion 23

3 UML modelling and detailed design **24**

3.1 Introduction 24
3.2 Use Case Diagram 24

3.3 Textual description of use cases 25
3.4 Sequence diagram: network flow prediction 25
3.5 Sequence diagram: blockchain storage 26
3.6 Simplified class diagram 26
3.7 Attack log data model 28
3.8 Smart contract design 28
3.9 Design security 28
3.10 Conclusion 29

4 Implementation, testing and results 31
4.1 Introduction 31
4.2 Development environment 31
4.2.1 Data Selection 31
4.3 Python implementation 32
4.4 Java implementation 33
4.5 Solidity implementation 33
4.6 Docker Execution 34
4.7 Tests carried out 34
4.8 Experimental results 35
4.9 Errors encountered and solutions 36
4.10 Discussion 36
4.11 Conclusion 37

General conclusion 38

References 39

List of Figures

- **Figure 1.1** – General IoT-Fog-Blockchain Architecture 10
- **Figure 1.2** – Example of a Slow HTTP-DoS attack 12
- **Figure 2.1** – Global Architecture of the Proposed Solution 18
- **Figure 2.2** – Project structure 21
- **Figure 3.1** – Use Case Diagram 24
- **Figure 3.2** – Sequence Diagram: Network Flow Prediction 26
- **Figure 3.3** – Sequence Diagram: Blockchain Storage 27
- **Figure 3.4** – Simplified Class Diagram 30
- **Figure 4.1** – Fog node execution 33
- **Figure 4.2** – IoT simulator results 33
- **Figure 4.3** – Ganache Transactions 35

List of Tables

- **Table 1.1** – Comparison between Cloud Computing and Fog Computing 12
- **Table 1.2** – Types of DoS and DDoS attacks 13
- **Table 2.1** – Software components of the system 15
- **Table 2.2** – Main functionalities 16
- **Table 2.3** – Functional requirements 16
- **Table 2.4** – Non-functional requirements 17

- **Table 2.5** – Technological choices 22
- **Table 3.1** – Textual description of use cases 25
- **Table 3.2** – Main classes and responsibilities 27
- **Table 4.1** – Experimental results 35
- **Table 4.2** – Encountered errors and applied solutions 36

General introduction

The Internet of Things, often referred to by the acronym IoT, has become one of the most important areas of modern computing. It enables physical objects, such as sensors, cameras, smart meters, industrial equipment and home gateways, to collect information and transmit it over a network. This development has led to the creation of smart cities, connected homes, automated industrial environments and remote monitoring systems. However, the increase in the number of connected devices has also expanded the attack surface available to cybercriminals. In an IoT environment, every connected device can become a potential entry point. The diversity of hardware, the weakness of certain configurations, the lack of regular updates and limited resources make protection more difficult. Denial-of-service (DoS) attacks are a major threat among these. They aim to render a service unavailable by consuming its resources. In the case of Slow HTTP-DoS attacks, the attacker does not necessarily seek to send a massive volume of data. Instead, they maintain numerous open HTTP connections by sending incomplete or very slow requests. This approach makes the attack more discreet and sometimes harder to distinguish from a com-
normal behaviour.

Traditional security solutions often rely on a centralised architecture in which data is sent to a cloud server for processing. Whilst this model can be effective in certain contexts, it has limitations in IoT environments. The constant transfer of data streams to the cloud increases latency, consumes bandwidth and can delay the response to an attack. Furthermore, when a large number of connected devices generate data simultaneously, centralised processing can become costly and less flexible.

Fog computing offers an interesting alternative. It involves bringing part of the processing closer to connected devices, using intermediate nodes known as fog nodes. These nodes can be placed near sensors or gateways and make decisions locally. In the context of security, a fog node can receive network traffic characteristics, analyse them quickly and trigger an action in the event of suspicious behaviour. This approach reduces latency, decreases the load sent to the cloud and allows for better adaptation to distributed environments.

Machine learning is also a useful tool for intrusion detection. By using measurable characteristics, such as the duration of a data stream, packet arrival intervals or packet size statistics, a model can learn to distinguish between normal traffic and traffic associated with an attack. In this project, a pre-trained model is loaded by a Flask fog node. The IoT simulator sends the twelve expected features, in the same order as that used during training, to avoid inconsistencies between the model, the scaler and the input data.

Detection alone is not always enough. In a security context, it is also important to retain reliable evidence of detected events. An attack log stored locally can be altered, deleted or disputed. To strengthen the integrity of the evidence, this project uses a local Ethereum blockchain. The blockchain does not store the full network data, as this can be voluminous, slow to write and sometimes sensitive. It stores only the SHA-256 hash of each attack log. Thus, if a log needs to be verified later, one simply needs to recalculate its hash and compare it to the value stored in the smart contract.

The overall objective of this thesis is therefore to design and implement a distributed prototype capable of simulating IoT traffic, detecting Slow HTTP-DoS attacks at the fog level using machine learning, logging detected attacks and storing evidence

on a local blockchain. The solution combines a Python component for simulation and detection, a Java component for blockchain logging, a Solidity smart contract for storing fingerprints, and Docker to facilitate the automation of the IoT-Fog component.

This thesis is organised into four chapters. The first chapter presents the general context, notably the IoT, Slow HTTP-DoS attacks, Fog Computing and blockchain. The second chapter is devoted to the requirements analysis and the design of the proposed solution. The third chapter presents the UML modelling and detailed design. The fourth chapter describes the implementation, the tests carried out, the results obtained, the errors encountered and the limitations of the prototype. Finally, a general conclusion summarises the work accomplished and suggests avenues for improvement.

Chapter 1

General context and problem statement

1.1 1.1 Introduction

This first chapter introduces the concepts necessary for understanding the project. It begins by presenting the Internet of Things and the constraints that characterise connected devices. It then describes denial-of-service attacks, in particular Slow HTTP-DoS attacks. The chapter also discusses Fog Computing as an approach to distributed processing close to the devices, followed by blockchain as a mechanism for proving integrity. These elements lead to the formulation of the problem statement and the project's objectives.

1.2 1.2 Internet of Things

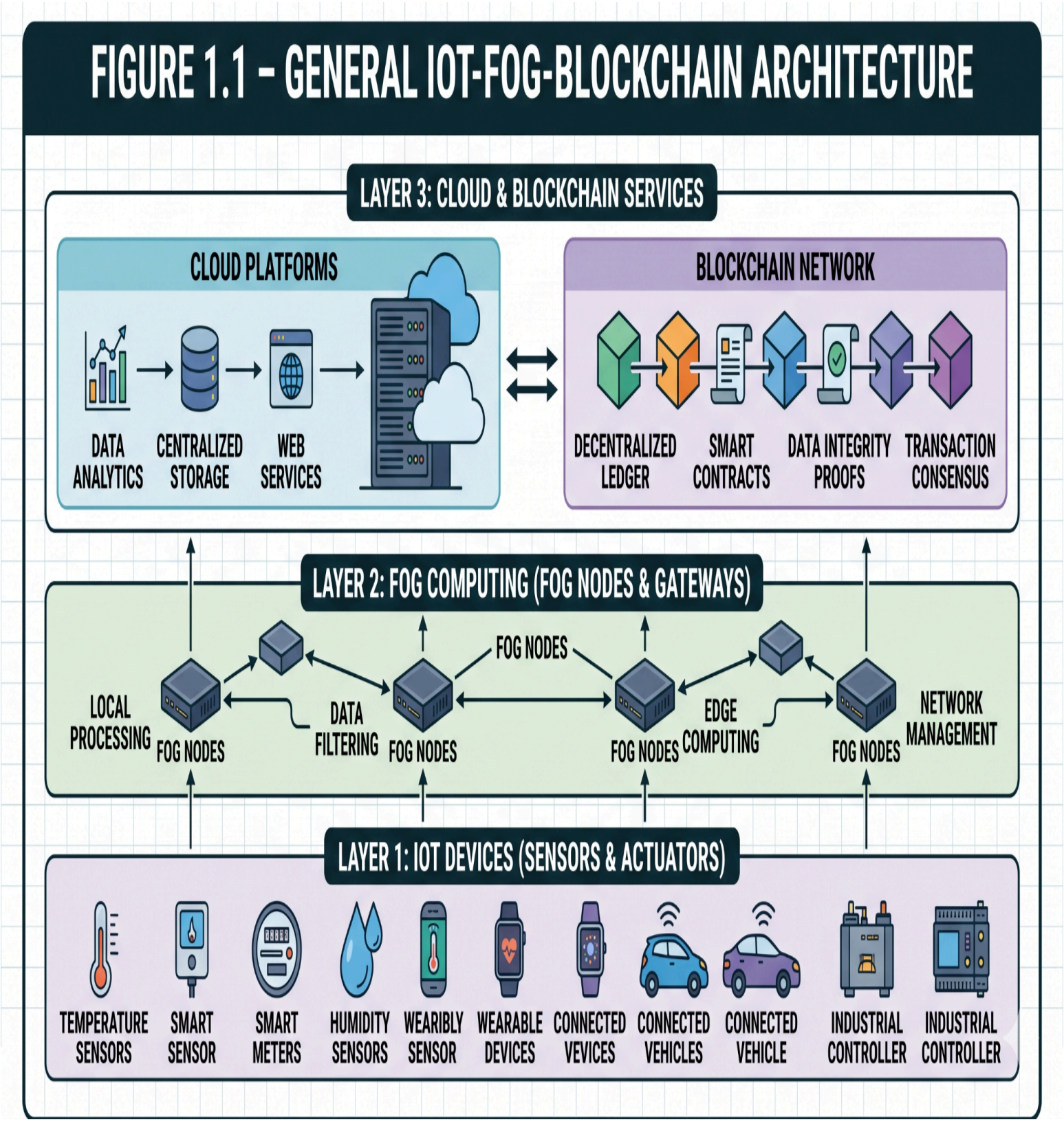
The Internet of Things refers to a collection of physical objects capable of collecting, processing and transmitting data via a network [1]. These objects can include temperature sensors, cameras, smart meters, medical devices, industrial equipment or home automation gateways. Their role is to transform a physical environment into a digital environment that can be monitored and controlled [1].

An IoT system is generally composed of several layers [2]. The first layer contains the sensors and actuators that interact directly with the real world. The second layer comprises gateways and communication equipment, responsible for transmitting data to application services. The third layer includes servers, cloud platforms or fog systems that analyse the data and provide services to users [2].

Smart environments, such as connected homes, smart cities or industrial systems, rely on this ability to interconnect numerous objects [1]. A sensor can measure data, a gateway can transmit it, a service can analyse it, and then an action can be triggered automatically. This automation brings significant benefits, but it also requires appropriate protection [3].

Connected devices are often characterised by limited resources [1]. They may have underpowered processors, limited memory, short battery life and variable bandwidth. These constraints prevent the use of overly complex security mechanisms [8]. For example, a small sensor cannot always run a complex traffic analysis algorithm or store a large volume of logs. It therefore becomes necessary to distribute security tasks across devices, gateways and fog nodes [2], [11].

Figure 1.1 – General IoT-Fog-Blockchain architecture



1.3 DoS and Slow HTTP-DoS attacks

A denial-of-service attack, known as a DoS attack, aims to make a service unavailable to its legitimate users. The attacker seeks to overwhelm a critical resource such as bandwidth, memory, the processor, the number of connections or server queues [4]. When the attack originates from multiple compromised machines, it is referred to as a distributed denial-of-service attack, or DDoS [10].

DoS and DDoS attacks can take many forms. Some rely on a large volume of packets, whilst others exploit weaknesses in protocols or configurations. In an IoT system, these attacks are particularly dangerous, as they can disrupt monitoring services, prevent the transmission of information or block critical equipment [7]. The unavailability of a service in a connected environment can have significant consequences for business continuity [10].

Slow HTTP-DoS attacks differ from traditional volumetric attacks [4]. Instead of sending a massive amount of traffic, the attacker opens multiple HTTP connections and keeps them active for as long as possible. They send request headers or bodies very slowly, sometimes one byte at a time [4]. The server then keeps the connections open whilst waiting for the request to complete. On a large scale, this behaviour consumes available resources and prevents new legitimate connections [7].

This attack is difficult to detect, as the traffic may resemble normal behaviour. Each connection may appear valid, and the overall volume is not always very high [4]. Filtering tools based solely on throughput may therefore fail to identify the attack [12]. It is necessary to observe finer details, such as the duration of data flows, arrival intervals, the number of packets, or packet length statistics [3], [12].

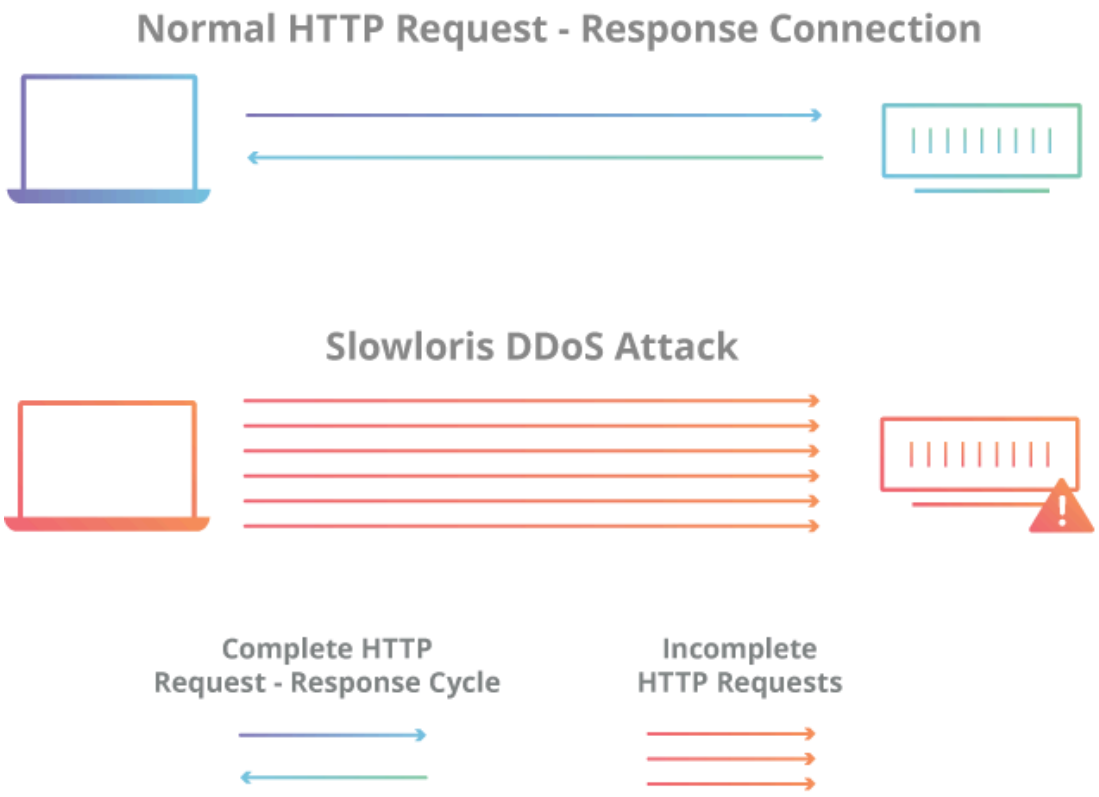


Figure 1.2 – Example of a Slow HTTP-DoS attack

Table 1.1 – Comparison between Cloud Computing and Fog Computing

Criterion	**Cloud Computing**	**Fog Computing**
Localisation of processes	Remote data centres	Nodes close to connected devices
Latency	Higher depending on network distance	Lower thanks to local processing
Bandwidth	High consumption when all data is sent to the cloud	Reduction in upstream traffic through intermediate processing
Security decision	Centralised decision	Distributed or semi-decision-making
Adapted to the IoT	Suitable for massive storage and global analytics	Suitable for rapid response and local monitoring

Table 1.2 – Types of DoS and DDoS attacks

Type of attack	Principle	Main impact
Volumetric	Massive traffic sent to the target	Bandwidth saturation
Protocol-based	Exploitation of behaviour of network protocols	Consumption of network or system resources
Application	Targeting a specific application service	Excessive load on the server or application
Slow HTTP DoS	HTTP connections kept open with slow or incomplete requests	Exhaustion of connection limits and server resources

1.4 Fog Computing

Fog Computing is a distributed architecture model that places computing, storage and communication resources between connected devices and the cloud [2]. Unlike a purely centralised architecture, the fog allows part of the processing to be carried out close to the data sources [2]. A fog node can be a local gateway, a mini-server, an industrial computer or network equipment with computing capabilities.

In the field of IoT security, Fog Computing offers several advantages [9]. It reduces latency, as data does not need to travel across the entire network to be analysed [2]. It reduces the load sent to the cloud, as only useful or summarised information needs to be transmitted. It also enables local decision-making, which is useful when an attack needs to be detected quickly in order to limit its impact [11].

Fog computing is also well-suited to distributed environments. Multiple fog nodes can be deployed across different areas, each responsible for a group of connected devices [2]. This architecture improves scalability and prevents a single central point from bearing the entire detection load [11]. In our prototype, the fog nodes are represented by Flask servers capable of receiving data streams, loading a model and logging detected attacks.

.

1.5 Blockchain and data integrity

Blockchain is a distributed ledger technology in which information is organised into blocks linked together using cryptographic mechanisms [6]. Each block contains transactions and a reference to the previous block. This structure makes it difficult to alter a record, as any change would have to be reflected in the subsequent blocks [6].

Ethereum is a blockchain platform that enables the execution of smart contracts [5]. A smart contract is a programme deployed on the blockchain and executed deterministically by the network's nodes [5]. Solidity is the language used to write these contracts [13]. Ganache, meanwhile, allows you to create a local Ethereum blockchain for testing purposes. It provides accounts, test private keys, an RPC server and an environment with no real-world costs.

In this project, the blockchain plays a specific role: it stores the SHA-256 hashes of attack logs. It does not store complete network traffic or raw log files [10]. This decision is important for

several reasons. Firstly, network data can be voluminous and unsuitable for blockchain storage [10]. Secondly, writing to the blockchain is slower than writing locally. Furthermore, logs may contain sensitive information, such as addresses or traffic details. Finally, a cryptographic fingerprint is sufficient to verify the integrity of a log [10].

The verification process is simple. When a log is generated, its hash is calculated and stored in the smart contract [13]. Later, if one wishes to prove that this log has not been altered, its hash is recalculated from the original line and the result is compared with the value stored on the blockchain. If the two values are identical, the log is considered intact. If they differ, this indicates that the content has been altered.

1.6 Challenges

IoT-Fog environments require security mechanisms capable of quickly detecting attacks whilst respecting the limitations of connected devices. Slow HTTP-DoS attacks are difficult to identify using simple methods, as they utilise slow connections that may resemble legitimate traffic. Furthermore, when an attack is detected, reliable evidence must be preserved without exposing sensitive data.

The challenge of this project can be formulated as follows: *How can we design and implement a distributed IoT-Fog system capable of detecting Slow HTTP-DoS attacks using machine learning, whilst preserving the integrity of logs of detected attacks by storing hashes on a local Ethereum blockchain?*

1.7 Project objectives

The main objectives of this project are as follows:

- to simulate IoT traffic using CSV datasets;
- to use a fog node to receive and classify traffic;
- detect Slow HTTP-DoS attacks using machine learning;
- generate attack logs in JSON Lines format;
- calculate the SHA-256 hash of each attack log;
- store the hashes in an Ethereum smart contract;
- demonstrate local storage using Ganache;
- set up Docker automation for a distributed fog demonstration.

1.8 Conclusion

This chapter has presented the theoretical foundations of the project. The IoT offers numerous possibilities, but remains vulnerable to denial-of-service attacks. Fog Computing brings the detection of connected devices closer to the edge, whilst blockchain can strengthen the integrity of evidence. These concepts form the basis of the solution proposed in the following chapter.

Chapter 2

Analysis and design of the proposed solution

2.1 Introduction

This chapter presents the requirements analysis and the design of the proposed solution. It describes the system's stakeholders, the functional and non-functional requirements, the overall architecture, the technical components, the operational workflow and the technological choices. The aim is to translate the security challenge into a clear and feasible architecture.

2.2 2.2 Needs analysis

The system under development must be capable of simulating network traffic, transmitting it to fog nodes, classifying each data flow, logging attacks and maintaining proof of integrity on the blockchain [10]. It must be sufficiently modular to separate the simulation component, the detection component, the local logging component and the blockchain component [2]. This separation facilitates maintenance, testing and future improvements.

2.2.1 2.2.1 System components

The system actors are the entities that interact directly or indirectly with the prototype. In this project, the main human actor is the administrator or student demonstrator, who launches the components, checks the results and analyses the logs. The other actors are software components, each playing a specific role in the detection chain.

Table 2.1 – System software components

Actor / component	**Role in the system**
IoT Simulator	Reads the CSV data and sends the twelve flow characteristics to the fog node.
Fog Node	Receives requests, applies the scaler, queries the model and writes attack logs.
Machine Learning Model	Classifies flows as NORMAL or ATTACK based on the selected characteristics.
Java Blockchain Logger	Reads the JSON logs, calculates the SHA-256 hashes and sends them to the smart contract.
Ganache Blockchain	Provides a local Ethereum blockchain for testing without incurring any real costs.
Smart Contract	Stores ID-hash pairs and allows you to view the number of records.
Administrator / student	Runs tests, checks outputs, consults Ganache and prepares the demonstration.

2.2.2 Functional requirements

The functional requirements describe the actions that the system must be capable of performing. They cover the entire chain, from data stream generation to verification of blockchain storage.

Table 2.2 – Key features

No.	Functionality
F1	Send network flows from the IoT simulator to one or more fog nodes.
F2	Receive data feeds via a Flask API that exposes a prediction endpoint.
F3	Load the classification model and scaler from the `models`.
F4	Check that all twelve features are present and in the correct order.
F5	Predict the class of the stream: NORMAL or ATTACK.
F6	Write a JSON log when an attack is detected.
F7	Read the logs from the Java programme.
F8	Calculate an SHA-256 hash for each log line.
F9	Store hashes in batches within an Ethereum smart contract.
F10	Read the number of stored records to verify the result.
F11	Launch multiple fog nodes with Docker Compose.

Table 2.3 – Functional requirements

Code	Description
BF01	The simulator must read a CSV file containing normal traffic, attack traffic or a mixture of both.
BF02	The simulator must send only the twelve selected characteristics expected by the model.
BF03	The fog node must expose a `/health` endpoint to allow its status to be checked.
BF04	The fog node must expose a `/predict` endpoint capable of receiving a stream and returning a prediction.
BF05	The model and scaler must be loaded automatically when the Flask server starts up.
BF06	When an attack is detected, a JSON line must be added to the `attack_logs_fognode1.json` file.
BF07	The Java programme must read the log file and ignore empty lines.
BF08	The Java programme must calculate the SHA-256 hash of each log entry without modifying the original log.
BF09	The hashes must be sent to the smart contract in batches to reduce the number of transactions.
BF10	The system must display the total number of hashes stored on the blockchain after execution.

2.2.3 Non-functional requirements

Non-functional requirements concern the expected qualities of the system. They are essential to ensure that the prototype is not only functional, but also understandable, reproducible and adaptable.

Table 2.4 – Non-functional requirements

Requirement	Explanation
Security	The system must limit the exposure of sensitive data by not storing raw logs on the blockchain.
Integrity	SHA-256 hashes must enable verification that local logs have not been modified.
Scalability	The architecture must be capable of being scaled to multiple fog nodes and multiple log files.
Maintainability	Components must be separated to facilitate error correction and project evolution.
Reproducibility	Commands, dependencies and paths must be documented to enable the project to be restarted at a later date.
Performance	The system must handle a reasonable number of flows without excessive slowdown during simulation.
Modularity	Blockchain simulation, detection and logging must remain independent.
Portability	Docker must facilitate the deployment of the fog and simulator components in different environments.

2.3 Overall architecture

The proposed architecture is based on three complementary layers [2]. The first is the IoT simulation layer, which generates network traffic from CSV files. The second is the fog detection layer, which receives the traffic, applies the scaler, queries the model and produces the logs [11]. The third is the blockchain integrity layer, which transforms the detected logs into cryptographic proofs stored in a smart contract [10].

This architecture clearly separates responsibilities. The simulator does not contain the classification logic. The fog node does not directly contain the blockchain storage logic. The Java programme does not alter the model's decisions, but intervenes after the logs have been generated to ensure their traceability [14]. This separation makes the system simpler to test and easier to explain in an academic demonstration.

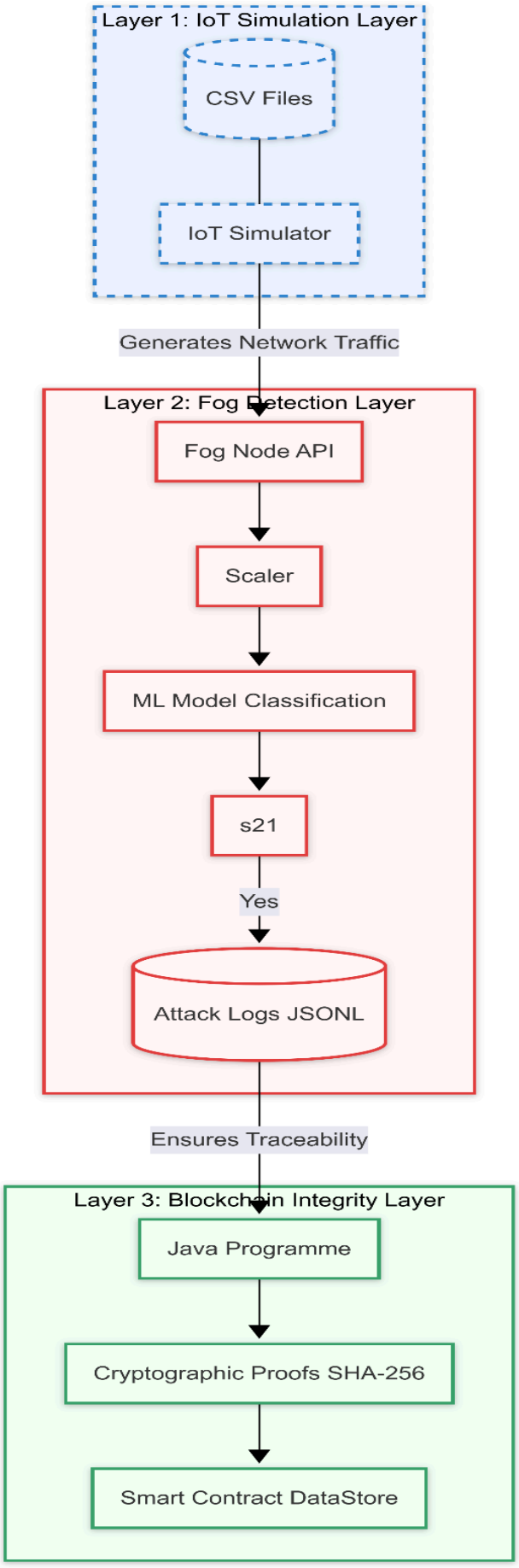


Figure 2.1 – Overall architecture of the proposed system

2.4 Description of components

2.4.1 IoT Simulator

The IoT simulator is implemented in Python in the file src/iot_simulator.py. Its role is to read a CSV dataset, extract the necessary characteristics and send each stream to a fog node using HTTP requests [11]. It can use a normal sample, an attack sample or a mixed sample.

The simulator plays an important role in prototype validation, as it allows an IoT environment to be replicated without the need for a large number of physical devices [7]. It also enables the calculation of metrics such as accuracy, precision, recall and F1-score when the actual labels are available in the CSV file [3].

2.4.2 Fog Node Flask API

The fog node is a Flask server located in src/fog_node.py. It exposes a /health route and a /predict route [2]. On startup, it loads the slow_http_model.pkl model and the scaler.pkl scaler [15]. When a stream arrives, it checks the features, applies normalisation and asks the model to produce a classification [8].

If the traffic is detected as an attack, the fog node writes a log in JSON Lines format to the logs folder. This log contains information useful for auditing, such as the timestamp, the name of the fog node, the source and destination addresses, the probability, and the list of features used [11].

2.4.3 Machine Learning Model

The machine learning model was trained offline in a notebook. It is based on a selection of twelve network features [8]. This selection must remain consistent across the notebook, the scaler, the model, the fog node and the simulator [3]. A significant error encountered during the project was the message ‘Expected 12 features, got 15’. This was caused by a discrepancy between the number of features sent and the number expected by the model. To resolve this issue, the final list of features was fixed and reused throughout the system. This rule is essential: even if the column names are correct, a change in order or number can result in an error or an inconsistent prediction..

2.4.4 Attack Log JSON

The attack_logs_fognode1.json file acts as the link between fog detection and blockchain logging. The JSON Lines format means that each line of the file is an independent JSON object. This choice makes it easier for the Java programme to read the file line by line, without having to load a large, complete JSON file into memory. An attack log is not stored directly on the blockchain [10]. It remains locally, in the logs folder. Only its hash is sent to the smart contract. The JSON file can therefore retain details useful for analysis, whilst the blockchain retains a compact and verifiable record [10].

2.4.5 Java Blockchain Logger

The Java programme is located in the NetBeans EthereumProject. Its main file is EthereumProject.java. It connects to Ganache via the URL http://127.0.0.1:7545 deploys the smart contract or uses the compiled bytecode, reads the JSON file, calculates the SHA-256 hash of each line, and stores the hashes in the smart contract in batches [10]. The use of Java is justified by the availability of Web3j, a library that allows interaction with Ethereum from a Java programme. In this project, Web3j is used directly with FunctionEncoder, FunctionReturnDecoder and RawTransactionManager, which avoids the need to generate a wrapper using the Web3j CLI..

2.4.6 Solidity DataStore Smart Contract

The DataStore smart contract contains a simple structure consisting of an identifier and a value [13]. The identifier corresponds to the attack log number and the value corresponds to the SHA-256 hash. The contract provides an addBatch function to store multiple records, a getCount function to retrieve the total count, and a getData function to read a record [13].

2.4.7 Ganache

Ganache is used as a local Ethereum blockchain [5]. It allows transactions to be tested without incurring any real costs and without any risk. It provides test accounts with local private keys. These keys must never be confused with real Ethereum wallets. In this project, Ganache is used solely to demonstrate how hash storage works.

2.4.8 Docker and Docker Compose

Docker enables multiple services to be launched in a reproducible manner. As the project currently stands, Docker automates the Python component, namely the fog nodes and the IoT simulator. This automation facilitates distributed demonstrations involving multiple ports and multiple nodes. The Java component works correctly when run manually using NetBeans, Maven and Ganache, but it has yet to be integrated into the full Docker automation workflow.

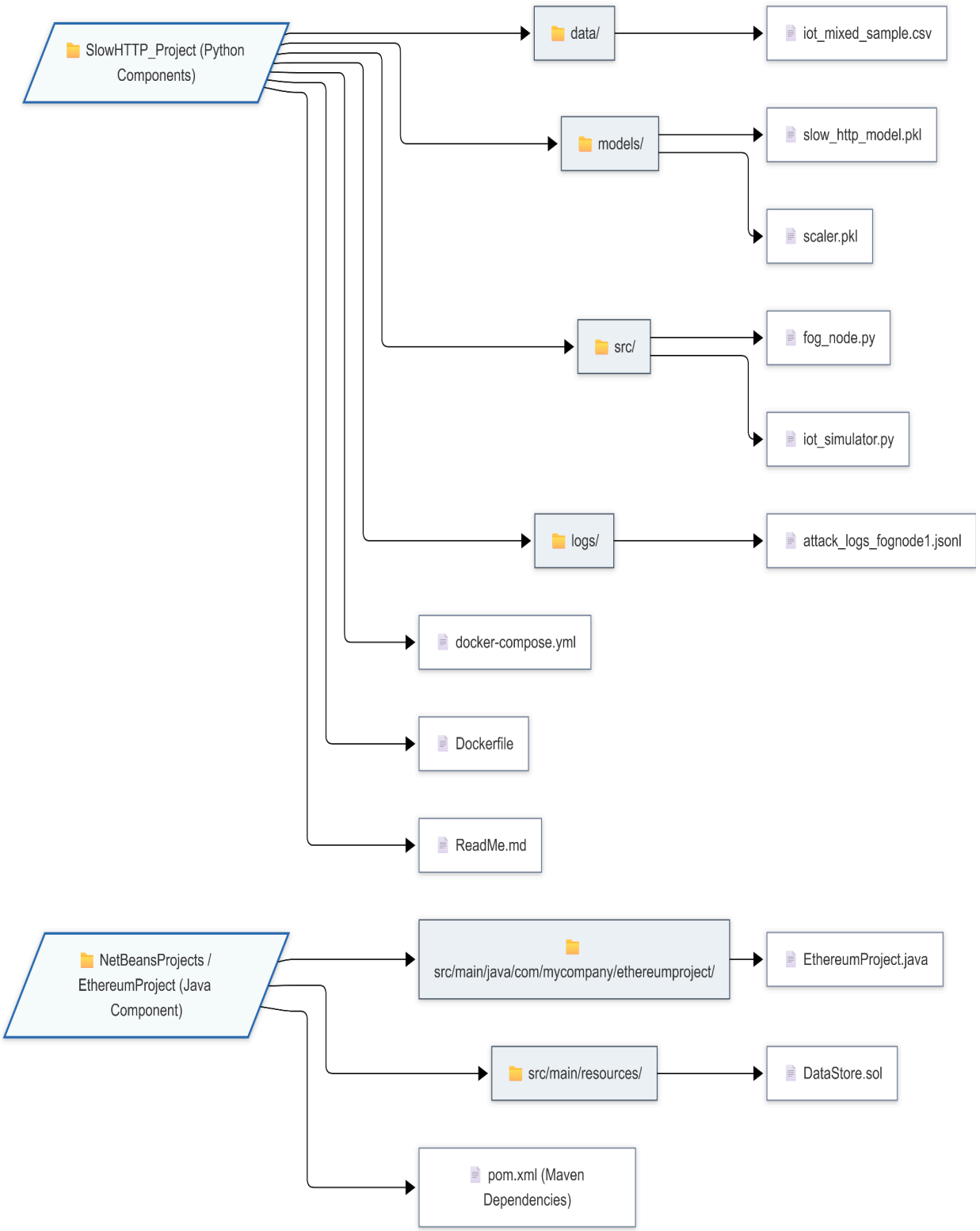


Figure 2.2 – Project structure

2.5 Workflow

The overall system workflow can be summarised in ten steps. Firstly, the simulator reads a CSV file. Secondly, it extracts the twelve expected features. Thirdly, it sends the data to the /predict endpoint of the fog node. Fourthly, the fog node applies the scaler. Fifthly, the model predicts the flow class [11]. Sixthly, if the class is ATTACK, the fog node adds a line to the JSON file. Seventhly, the Java programme reads this file. Eighthly, it calculates the SHA-256 hash of each line. Ninthly, it deploys or uses the DataStore smart contract. Tenthly, it stores the hashes in batches and reads the total count to verify the result [10].

This workflow adheres to the principle of separating data from proof. The logs remain available locally for analysis, whilst the blockchain retains only the fingerprint that proves their integrity [10]. Should a line be modified, even minimally, the recalculated hash will differ from the one stored, enabling the alteration to be detected [6].

2.6 Technological choices

Table 2.5 – Technological choices

Technology	**Role**	**Reason for selection**
Python	Detection and simulation	A simple language, rich in data and machine learning libraries.
Flask	REST API	Lightweight server, easy to configure for the `/health` and `/predict routes`.
scikit-learn	ML Model	Library suitable for classification and model saving.
pandas / numpy	Data manipulation	Reading CSV files, processing columns and preparing vectors.
joblib	Model loading	Efficient saving and loading of the scaler and the trained model.
Java	Blockchain logger	Robust integration with Web3j and Maven.
Maven	Dependency management	Java project build and library retrieval.
Web3j	Ethereum interaction	Sending transactions and calling smart contract functions.
Solidity	Smart contract	Defining storage logic on Ethereum.
Ganache	Local blockchain	Free, quick and reliable tests.
Docker	Automation	Reproducibility and deployment of multiple fog nodes.

2.7 Status of Docker automation

Docker automation is a key part of the project, as it allows multiple components to be launched without repeating all the manual commands. Currently, the Python part works with Docker: the fog nodes and the simulator can be launched via Docker Compose, and the system generates attack logs [11].

However, the Java blockchain component is described as a separate component. It runs manually using NetBeans, Maven and Ganache. The Java programme connects to the local blockchain, deploys the smart contract, reads the logs, calculates the hashes and stores the values [10]. The final improvement involves integrating this Java logger into Docker, for example as a separate service that waits for the logs to be generated before initiating blockchain storage.

This situation is acceptable for a university prototype, as it demonstrates that each component is functioning correctly. It also makes it possible to distinguish between work that has already been validated and the remaining integration work. For the demonstration, Docker can be used for the fog nodes, followed by NetBeans for the blockchain storage, whilst explaining that full automation is a future development prospect.

2.8 Conclusion

This chapter has presented the proposed solution in the form of a three-layer modular architecture: IoT simulation, fog detection and blockchain integrity. The functional and non-functional requirements have been detailed, the components explained and the technological choices justified. The following chapter presents the UML modelling and the detailed design of the system.

Chapter 3

UML modelling and detailed design

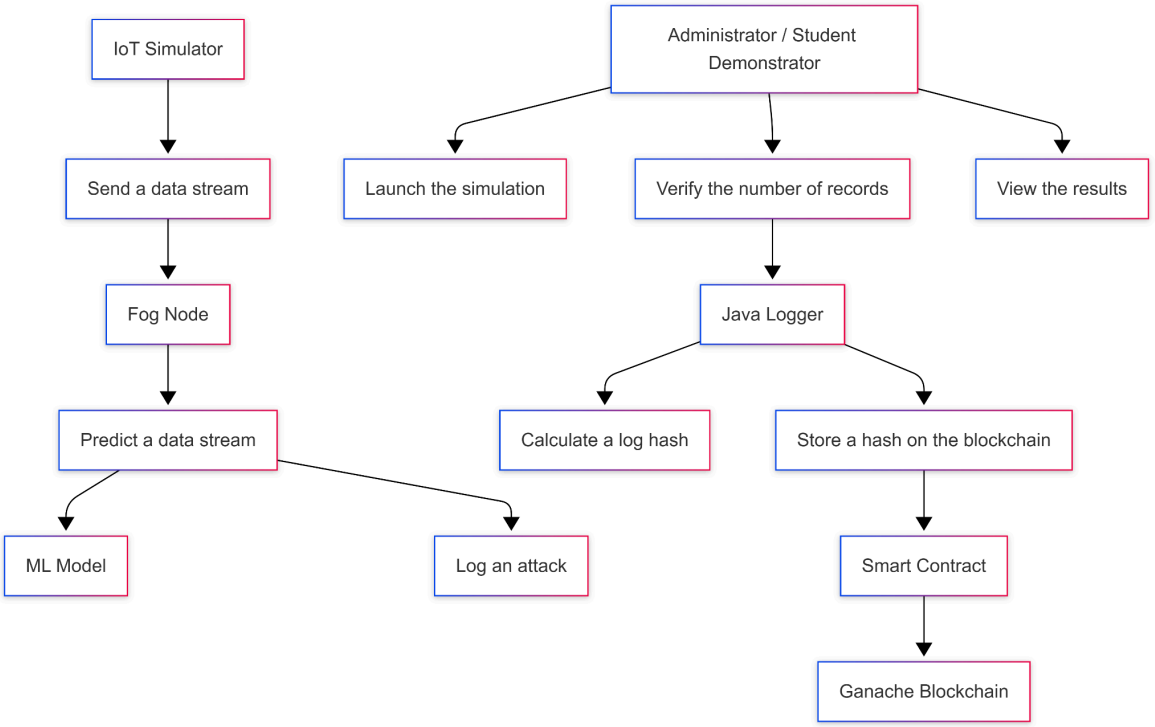
3.1 Introduction

UML modelling allows the system to be represented in the form of diagrams that are understandable to developers, supervisors and evaluators. It does not replace the code, but it makes it easier to understand the interactions, responsibilities and data being handled. In this chapter, we present use cases, sequence diagrams, a simplified class diagram, the attack log data model and the smart contract design [13].

3.2 Use Case Diagram

The use case diagram presents the system's visible functionalities. The main human actor is the administrator or the student demonstrator. The other technical actors are involved in the internal workings: IoT simulator, fog node, ML model, Java logger, Ganache blockchain and smart contract [10]. The main use cases are: launching the simulation, sending a data stream, predicting a data stream, logging an attack, calculating a log hash, storing a hash on the blockchain, verifying the number of records, and viewing the results. The diagram illustrates that the system is not a monolithic application, but a chain of cooperating components [2], [11].

Figure 3.1 – Use case diagram



3.3 Textual description of use cases

Table 3.1 – Textual description of use cases

Use Use Case	Actor	Precondition	Main scenario	Postcondition
Run the simulation	Administrator	The CSV file exists and the fog node is running.	The user defines the dataset and the number of streams, then starts the simulator.	The streams are sent to the fog node.
Send a stream	IoT Simulator	The twelve characteristics are available.	The simulator constructs an HTTP request containing the stream's characteristics.	The fog node receives the request.
Predict a stream	Fog Node / ML Model	The model and scaler are loaded.	The fog node normalises the values and calls the classification model.	A NORMAL class or ATTACK is returned.
Log an attack	Fog Node	The prediction is ATTACK.	The fog node creates a JSON object containing the relevant information and adds it to the JSON file.	A new log entry is recorded.
Hashing a log	Java Blockchain Logger	The JSON file contains at least one line.	The programme reads each line and applies SHA-256.	A list of hashes is prepared.
Store on the blockchain	Java Blockchain Logger / Smart Contract	Ganache is launched and the contract is available.	Java calls `addBatch` with the identifiers and hashes.	The hashes are recorded in the contract.
Check the number	Administrator / Java Logger	Some transactions have been sent.	Java calls `getCount` and displays the result.	The number of stored logs is confirmed.
View the results	Administrator	The tests are complete.	The user observes the simulator output, the logs, the Java console and Ganache.	The demonstration can be analysed.

3.4 Sequence diagram: network flow prediction

The first dynamic scenario involves predicting network traffic. The IoT simulator extracts the twelve characteristics and sends them to the fog node. The fog node transmits these values to the scaler for normalisation, then to the ML model to obtain a prediction [8]. If the result is NORMAL, the response is returned without logging an attack. If the result is ATTACK, the fog node writes a line to the JSON file before responding to the simulator [11].This sequence demonstrates that the security decision is made locally at the fog level [9]. It also shows that logging is conditional: only detected attacks generate attack logs. This approach avoids generating files that are too large for normal traffic.

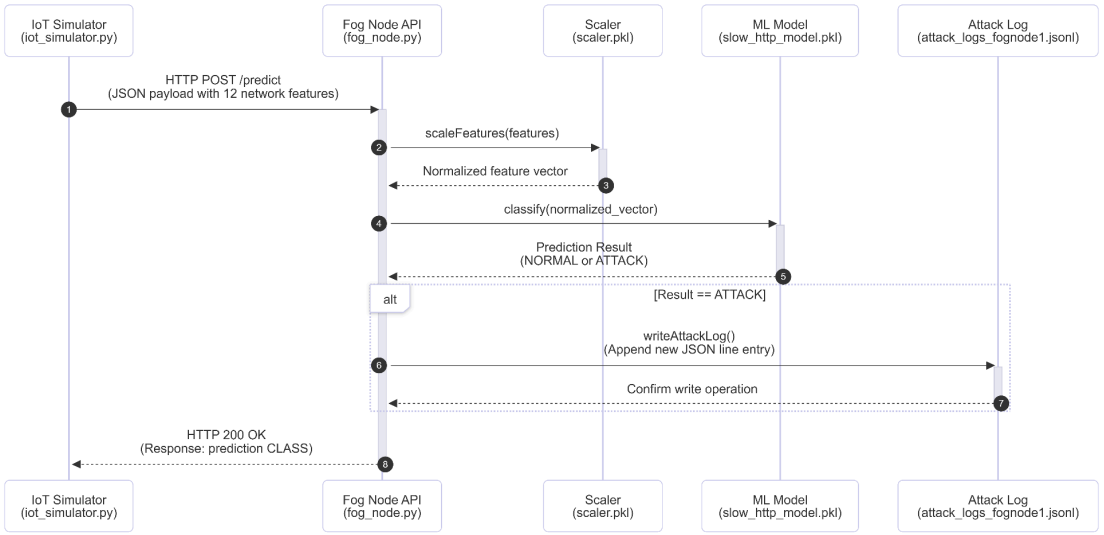


Figure 3.2 – Sequence diagram: prediction of a network flow

3.5 Sequence diagram: blockchain storage

The second dynamic scenario concerns the storage of integrity proofs [10]. The Java programme reads the attack_logs_fognode1.json file line by line. For each line, it calculates an SHA-256 hash. It then prepares arrays of identifiers and values. Web3j encodes the call to the addBatch function, signs the transaction with a local Ganache private key and sends it to the smart contract. Ganache executes the transaction and modifies the state of the local blockchain [5].

After sending the batches, the Java programme calls getCount. If the result matches the number of logs processed, this confirms that the hashes have been stored. It can also call getData(0) to display the first record and verify that the ID and hash are accessible [13].

Figure 3.3 – Sequence diagram: storing a hash on the blockchain

3.6 Simplified class diagram

The simplified class diagram illustrates the main software entities of the prototype. The IoTSimulator class contains methods for reading CSV files and sending data streams [11]. The FogNode class manages the Flask API, the health route, the prediction route and log writing [2]. The MLModel class represents the model and the scaler [8]. The AttackLog class represents the structure of a JSON line.

The blockchain component can be modelled using BlockchainLogger, HashService and DataStoreContract. HashService calculates SHA-256 hashes. BlockchainLogger reads the logs, prepares batches and communicates with Ethereum via Web3j [10]. DataStoreContract represents the functions exposed by the smart contract [13].

Figure 3.4 – Simplified class diagram

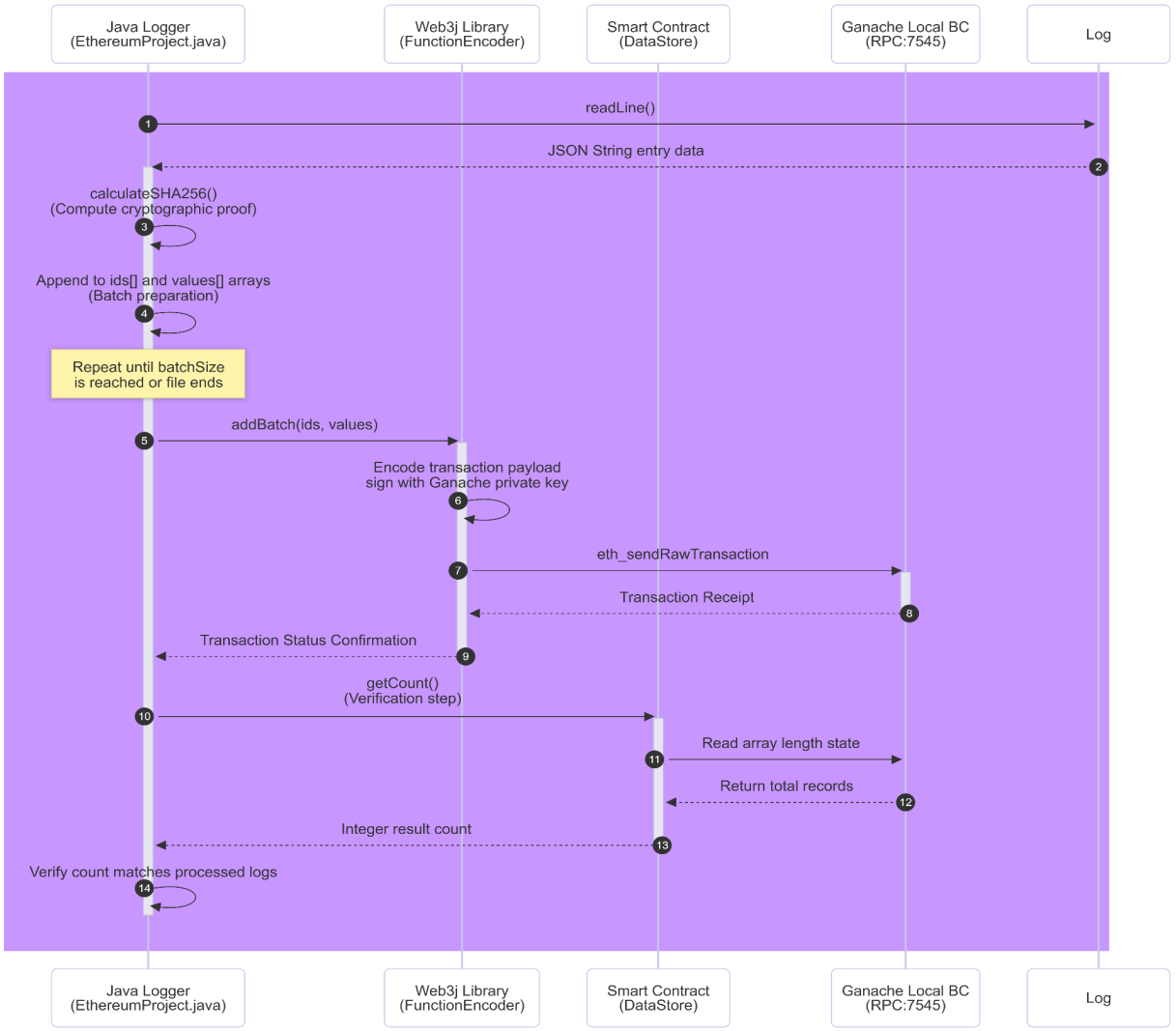


Figure 3.3 – Sequence diagram: storing a hash on the blockchain

Table 3.2 – Main classes and responsibilities

Class	Possible attributes	Main methods
IoT Simulator	dataset Path, numFlows, fog Nodes	readDataset(), buildPayload(), sendFlow(), printMetrics()
Fog Node	nodeName, modelPath, scaler-Path, logDir	health(), predict(), writeAttack-Log()
Model	selectedFeatures, model, scaler	loadModel(), scaleFeatures(), classify()
Attack Log	timestamp, node, sourceIp, destinationIp, probability, features	toJsonLine(), validate()
Hash Service	algorithm	sha256(), bytesToHex()

Fog Node	nodeName, modelPath, scaler-Path, logDir	health(), predict(), writeAttack-Log()
Blockchain Logger	ganache Url, privateKey, contractBytecode, batchSize	connect(), deployContract(), addBatch(), getCount()
DataStore Contract	dataset	addBatch(), getCount(), get-Data()

3.7 Attack log data model

The attack log uses the JSON Lines format. Each line is independent and represents a detected attack. This format is simple to generate from Python, easy to read from Java, and well-suited to incremental writing. It avoids the need to maintain a complete JSON array, which would be less practical when logs are added incrementally.

```
{
    "timestamp": "2026-05-06T14:30:00",
    "node": "FogNode1",
  "source_ip": "192.168.1.10",
    "dest_ip": "192.168.1.20",
    "probability": 0.99,
    "features": [120045, 450.1, 35.2, 900.0, 10.0, 0.23, 65.7,
  12.4,
      0.0, 1.1, 5000.0, 1000.0]
}
```

Listing 3.1 – Example of a JSON attack log entry

The fields may vary depending on the simulator version, but the principle remains the same: to retain locally the information useful for analysis and store only the fingerprint on the blockchain. The exact line is used as the input for the SHA-256 function. Any modification to the content, even a single character change, produces a different hash.

3.8 Smart contract design

The DataStore smart contract is deliberately simple in order to focus on the principle of integrity. It contains a Data structure consisting of id and value. The dataset variable is a public data array. The addBatch function takes two arrays, one for identifiers and the other for hashes. It checks that the two arrays are the same length, then adds the records to the array [13].

The getCount function returns the size of the array. It is used by Java to verify that the number of stored logs matches the expected number. The getData function allows a record to be read by its index. In this project, the stored value is always an SHA-256 hash and not the raw content of a log [10].

3.9 Security of the design

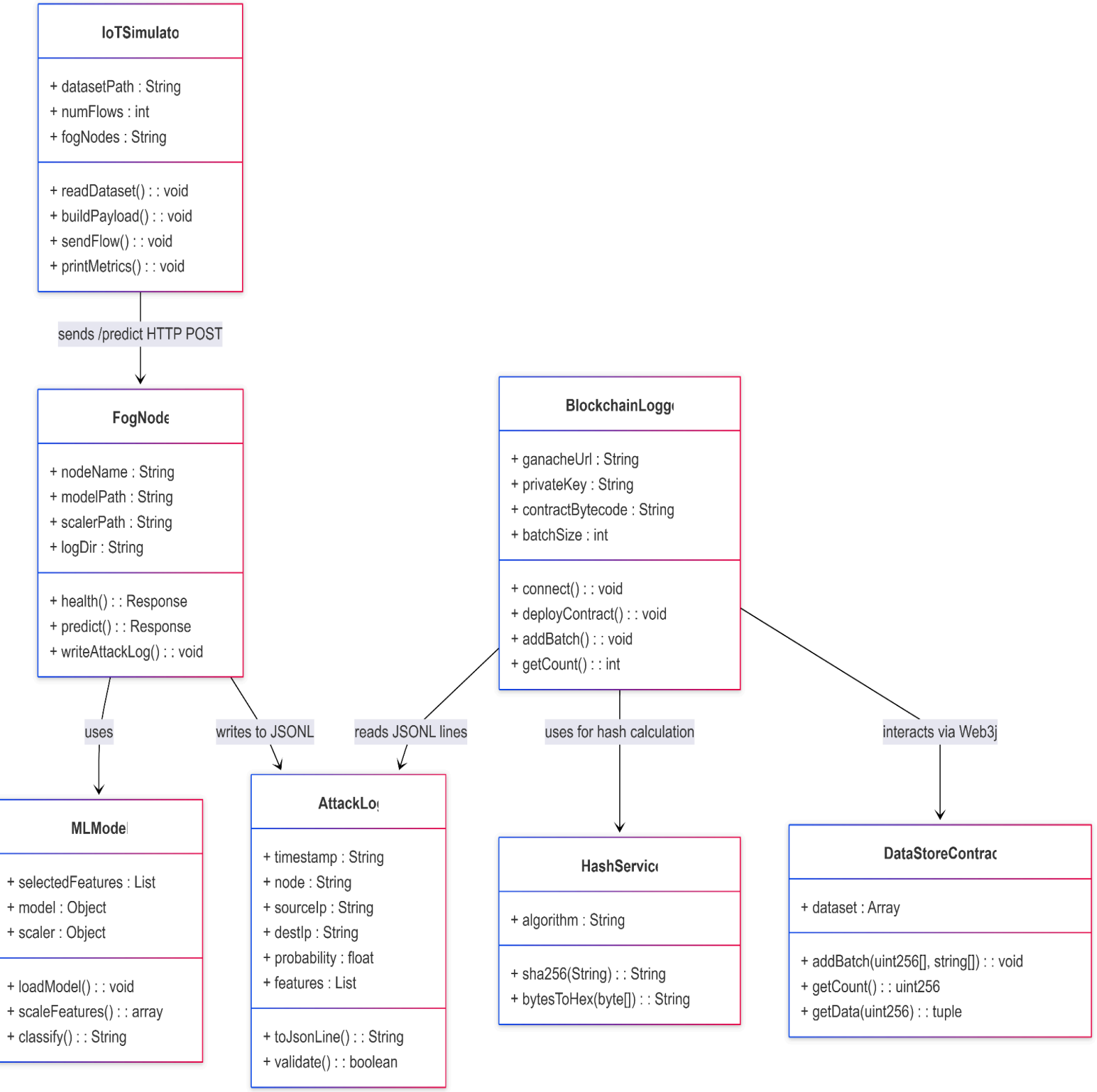
The proposed design adheres to several security principles. Firstly, the raw logs remain local. This limits the exposure of potentially sensitive data [2]. Secondly, the blockchain stores only cryptographic hashes. These hashes do not allow the original content of the log to be reconstructed, but they do allow its integrity to be verified [10]. Thirdly, the verification is independent of the programme that generated the log: one simply needs to recalculate the hash and compare it.

Ganache is used solely in a test environment. For a live deployment, it would be preferable to use a private Ethereum blockchain, controlled by the organisation, with authorised nodes [5]. It would also be useful to implement authentication between the fog nodes and the blockchain logger, to prevent unauthorized log files from being processed [1].

3.10 Conclusion

This chapter has presented a detailed model of the system. The use cases, sequences and classes demonstrate how the components work together to detect, log, hash and store evidence of attacks. The following chapter describes the practical implementation, testing and the results observed.

Figure 3.4 – Simplified class diagram



Chapter 4

Implementation, testing and results

4.1 Introduction

This chapter presents the practical implementation of the project. It describes the development environment, the Python, Java and Solidity components, the Docker commands, the tests carried out, the experimental results and the errors encountered [10], [11]. The aim is to show that the prototype is not merely a theoretical design, but a functional chain validated by testing.

4.2 Development environment

Development and testing were carried out primarily on Windows. The Python component uses Flask for the fog server, pandas and numpy for the data, scikit-learn and joblib for the model and scaler, and requests for sending streams from the simulator. The blockchain part uses Java, Maven, NetBeans, Web3j, Remix, Solidity 0.8.19 and Ganache v7.7.3.

The project Python is organised in the folder `SlowHTTP_Project`. The files important are the data in `data`, the templates in `models`, the scripts in `src`, the logs in `logs`, as well as the files Docker and the `ReadMe.md`. The Java project is located in `C:\Users\dell tac\Documents\NetBeansProjects\EthereumProject`, with the main file `src/main/java/com/mycompany/ethereumproject/EthereumProject.java`

4.2.1 Data Selection :

For the training of our Decision Tree model on the CICIDS2017 dataset (`Wednesday-workingHours` file), we made the technical choice to reduce the input data from over 80 initial features to **only 12**. This drastic reduction is a deliberate optimization decision, driven by three critical factors of our architecture:

1. **Elimination of noise and redundancy:** The majority of the features generated by CICFlowMeter are highly correlated with each other or irrelevant for slow application-layer attacks. By applying a Feature Selection method, we identified that adding variables beyond the top 12 brought no significant improvement to the detection accuracy, but instead increased the risk of overfitting.
2. **Strict constraints of Fog Computing:** Our Fog node (IoT gateway) has extremely limited computing power and memory capacities compared to a Cloud server. Each additional feature requires inspecting more packets in real-time. Limiting the model to only 12 features ensures critical response times (low latency) and preserves the node's energy and memory resources.
3. **Maintaining high performance:** As demonstrated by our experimental results (Table 4.1), this lightweight 12-feature model still manages to achieve a 100% precision and an F1-score of 93.08%. This specific subset therefore represents the perfect trade-off between processing efficiency and detection accuracy for Slow-HTTP-DoS requests (by

focusing strictly on vital variables such as `Flow Duration`, `Flow IAT`, and `TCP Flags`).

.

4.3 Python implementation

The `fog_node.py` file implements the fog node. On startup, it reads the environment variables that specify the path to the model, the path to the scaler, the logs directory, the node name and the port. It then loads the `slow_http_model.pkl` and `scaler.pkl` files. The `/health` route allows you to check that the server is operational. The
`/predict` receives the stream characteristics and returns a prediction.

The `iot_simulator.py` file reads the streams from a CSV dataset. For each row, it selects the twelve expected features and sends them to the configured fog node(s). It compares the response with the true class where available. Finally, it calculates the performance metrics, which allow the quality of the detection to be assessed.

The twelve features used are: Flow Duration, Flow IAT Mean, Flow IAT Std, Flow IAT Max, Flow IAT Min, Bwd Packets/s, Packet Length Mean, Packet Length Std, Active Std, Idle Std, Idle Max and Idle Min. This list must remain the same in all files. Correcting the error `'Expected 12 features, got 15'` was a crucial step, as it demonstrated the importance of consistency between training and execution.

```
  Actual NRM      FP = 0              TN = 150

PER-NODE BREAKDOWN
------------------------------------------------------------
  FogNode1: 56 flows | Acc=100.0% | Det=0% | Lat=28.0ms
  FogNode2: 58 flows | Acc=100.0% | Det=0% | Lat=28.2ms
  FogNode3: 36 flows | Acc=100.0% | Det=0% | Lat=28.0ms

============================================================
```

Figure 4.1 – Fog node execution

```
CONFIGURATION
------------------------------------------------------------
  Flows sent:     150
  Fog nodes:      3

PERFORMANCE
------------------------------------------------------------
  Duration:       4.22s
  Throughput:     35.6 flows/sec
  Avg latency:    28.0 ms

DETECTION METRICS
```

Figure 4.2 – IoT simulator results

4.4 Java implementation

The Java component is responsible for connecting to the blockchain. The `EthereumProject.java` programme begins by connecting to Ganache at the address `http://127.0.0.1:7545`. It uses a local private key provided by Ganache. This key is intended for testing purposes only and must never be replaced with a real wallet key.

Next, Java deploys the smart contract from the bytecode copied from Remix. After deployment, it reads the file `attack_logs_fognode1.json`. Each line is passed through an SHA-256 hashing function, and the hashes are then grouped into batches. Using batches reduces the number of transactions: for example, with 51 logs and a batch size of 10, this results in one deployment transaction and several storage transactions.

The programme verifies the result by calling `getCount`. If the contract returns 51 when 51 logs have been processed, this confirms that the values have indeed been added. Java can also read the first record to display its ID and hash.

4.5 Solidity implementation

The contract used must be compiled with Solidity 0.8.19. This version is chosen to avoid certain incompatibilities between the generated bytecode and the Ganache environment used during testing. Newer versions such as Solidity 0.8.20 or later may produce bytecode targeting a newer EVM version. If Ganache or its EVM configuration does not recognise certain opcodes, deployment may fail with an `'invalid opcode'` error. For this reason, the project retains `` `pragma solidity 0.8.19;` ``

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;
```

```
                string value;
        }

        Data[] public dataset;

        function addBatch(uint256[] memory ids, string[] memory
    values) public {
                require(ids.length == values.length, "Mismatched
        input");

                    for (uint256 i = 0; i < ids.length; i++) {
            dataset.push(Data(ids[i], values[i]));
                }
        }

        public view function getCount() returns (uint256) {
                return dataset.length;
        }

        function getData(uint256 index) public view returns (uint256,
    string memory) {
                Data memory item = dataset[index];
                return (item.id, item.value);
        }
}
```

4.6 Docker execution

Docker is used to automate the Python part of the project. The main commands are as follows:

```
docker compose build
docker compose up fog-node-1 fog-node-2 fog-node-3
docker compose run --rm iot-simulator
docker compose down
```

The health checks can be viewed via the following URLs: `http://localhost:5001/health`, `http://localhost:5002/health` and `http://localhost:5003/health`. This configuration demonstrates the ability to simulate multiple fog nodes. However, manual testing on Windows remains simpler for debugging, particularly when integrating with NetBeans, Maven and Ganache.

As things stand, Docker correctly automates the fog nodes and the IoT simulator. The system reaches the log generation and hashing stages. The Java logger works, but it needs to be integrated into the final automation as a Docker service or as a step triggered after the JSON file has been generated.

4.7 Tests carried out

Several tests were carried out to validate the prototype. The health check verifies that the fog node starts up and correctly loads the model and the scaler [2]. The simulator test verifies that the streams are sent and the results are displayed [11]. The feature compatibility test verifies that the system uses exactly the twelve selected features [3]. The logging test verifies that the detected attacks are written to the JSON file [11].

The Java part was tested separately using Maven and NetBeans. The Maven build confirmed that the Web3j and SLF4J dependencies are loaded correctly. The smart contract deployment test confirmed that Solidity 0.8.19 bytecode works with Ganache [5]. The hash storage test confirmed that the logs are read, hashed and sent to the contract [10]. Finally, the count verification test confirmed that the smart contract returns the expected number of records [13].

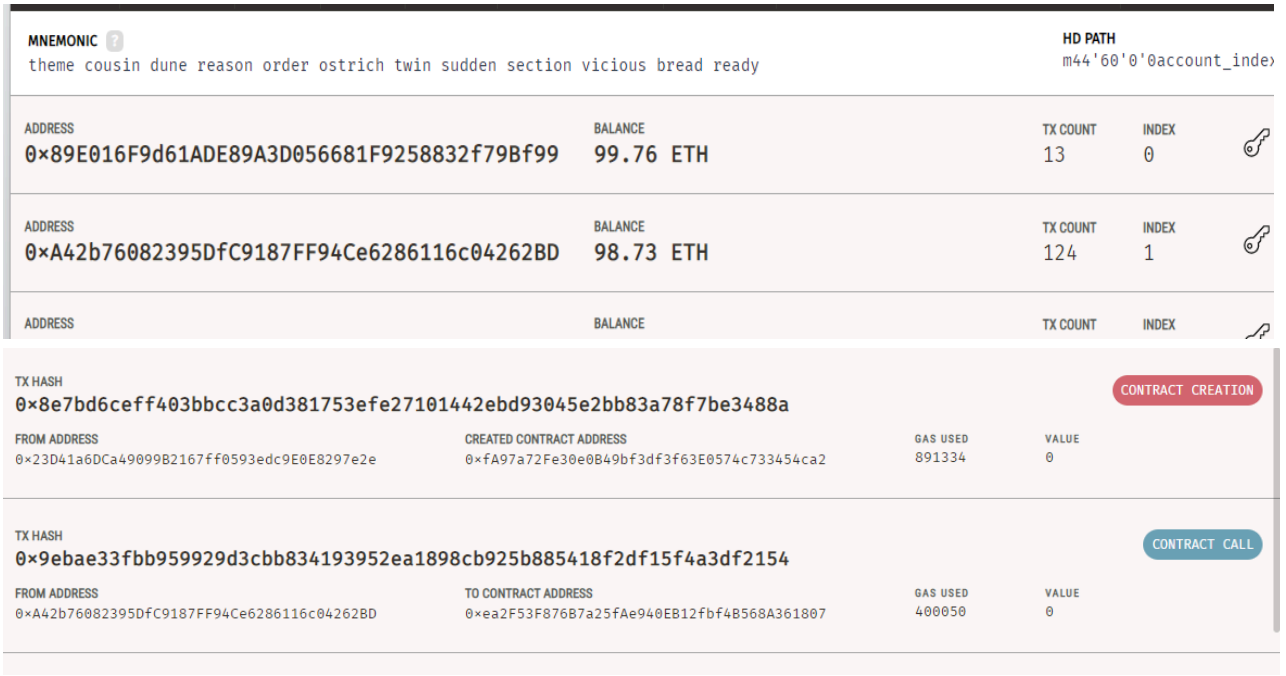


Figure 4.3 – Ganache Transactions

4.8 Experimental results

In a mixed simulation of 150 flows, the observed results are summarised in the following table.

Table 4.1 – Experimental results

Metric	**Value**
Total flows	150
Accuracy	92.67%
Précision	100.00%
Recall	87.06%
F1 score	93.08%
TP	74
FP	0
FN	11
TN	65

These results demonstrate the very high accuracy achieved by the Decision Tree algorithm implemented for this detection module [4], [12]. Chosen for its lightweight footprint—making it highly suitable for deployment at the resource-constrained Fog layer—the model ensured that no normal traffic was incorrectly classified as an attack, which explains the 100% precision [8]. This means that the alerts generated are highly reliable in this scenario. The recall is slightly lower, as 11 attacks were missed. This observation is important: it shows that the Decision Tree model can still be tuned to reduce false negatives [4].

A subsequent manual test generated the file attack_logs_fognode1.json, containing 51 attack logs [11]. These logs were prepared for hashing, and the hashes were then securely stored on the local blockchain using the Java programme [10]. This step successfully validates the transition from Machine Learning-based Fog detection to blockchain-backed integrity verification [10].

4.9 Errors encountered and solutions

Error	Cause	Solution
`Cannot open file fog_node.py`	The file is located in the `src` folder.	Use `python src\fog_node.py`.
`Expected 12 features, got 15`	The simulator sent a different number of features than expected.	Use the same 12 features in the notebook, the scaler, the model, ` `fog_node.py` ` and `iot_simulator.py`.
`class BlockchainTest is public...`	The name of the public class did not match the filename.	Use `public class EthereumProject` in `EthereumProject.java`.
`duplicate class BlockchainTest`	Two files contained the same public class.	Delete the duplicate class and keep a single correct file.
`EthSendTransaction cannot be converted to TransactionReceipt`	Web3j returns `EthSendTransaction` after `sendTransaction`.	Retrieve the transaction hash and then wait for the receipt using `PollingTransactionRe`
`exceeds block gas limit`	The requested gas limit exceeded the Ganache block limit.	Use a limit such as 5,000,000 or adjust the Ganache configuration.
`VM Exception: invalid opcode`	Bytecode compiled with an incompatible Solidity/EVM version.	Use Solidity 0.8.19 and, if necessary, the Paris EVM in Remix.

Table 4.2 – Errors encountered and solutions applied

4.10 Discussion

The prototype has several key strengths. It is modular, as each component has a clearly defined role [2]. It enables distributed detection through fog nodes [11]. It limits data exposure by not storing raw logs on the blockchain [10]. It provides a simple integrity check using SHA-256 hashes [10]. It is also reproducible, as the steps, commands and main paths are documented. The prototype also has its limitations. Ganache is a local test blockchain and does not represent a production network [5]. The smart contract is currently deployed with every Java execution, whereas a more advanced version could deploy the contract just once and reuse its address [13]. Docker integration is not yet complete for the Java component. The model still produces false negatives, which warrants future improvement [4].

4.11 Conclusion

This chapter has demonstrated that the prototype works through a series of tests. The Python component detects attacks and generates logs. The Java component reads these logs, calculates the hashes and stores them on Ganache. The results obtained confirm the feasibility of the proposed approach, whilst highlighting potential improvements.

General conclusion

This final-year project focused on the detection of Slow HTTP-DoS attacks in IoT-Fog systems using blockchain-based logging [10]. The aim was to design and build a prototype capable of simulating IoT traffic, detecting attacks at the fog node level, generating attack logs, and storing integrity proofs on a local Ethereum blockchain.

The prototype implemented establishes a complete chain: IoT simulator, Flask fog node API, machine learning model, JSON Lines log file, Java hashing programme, Solidity smart contract and Ganache blockchain [11]. This chain demonstrates that it is possible to distribute responsibilities across several complementary components. The simulated IoT devices do not perform heavy processing. The fog node makes security decisions close to the data sources [2]. The Java programme transforms the logs into SHA-256 hashes. The blockchain stores the integrity proofs [10].

The use of fog computing is particularly beneficial for IoT environments, as it reduces latency and limits the load sent to the cloud [2]. In the context of Slow HTTP-DoS attacks, this proximity is important: the faster the detection, the more the impact on services can be mitigated. Machine learning provides classification capabilities based on network flow characteristics. Experimental results show high accuracy and a satisfactory F1-score, whilst revealing the presence of false negatives that can be reduced through future work [8].

The blockchain has been used in a targeted manner. It does not store the full network data or raw logs [10]. It retains only the SHA-256 hashes. This approach meets the requirements for data volume, performance and confidentiality. It allows for subsequent verification that a local log has not been altered. The verification process is simple: retrieve the original log, recalculate its hash, and compare the resulting value with the one stored in the smart contracts [10].

The project also helped to resolve several practical difficulties. The feature mismatch error highlighted the importance of using exactly the same twelve features in the training notebook, the scaler, the model, the fog node and the simulator [3]. Errors related to Java and Web3j provided a better understanding of Ethereum transaction management. The invalid opcode issue highlighted the importance of compatibility between the Solidity version, the generated bytecode and the Ganache environment [13].

Several improvements could be considered. Firstly, the smart contract could be deployed once and its address reused. Secondly, the Java logger could be automated within Docker so that the entire pipeline is launched by a single command. Thirdly, REST communication could be added between the fog nodes and the Java logger. Fourthly, a dashboard could display attacks, metrics, hashes and transactions. Fifthly, the ML model could be improved to reduce false negatives [4]. Sixthly, authentication should secure exchanges between components. Seventhly, the fog nodes could be deployed on Raspberry Pi devices for a more realistic demonstration [2]. Eighthly, Ganache could be replaced by a private Ethereum network. Ninthly, the smart contract could emit events [13]. Finally, detection of duplicate hashes could prevent redundant records.

In conclusion, this project demonstrates a practical and educational architecture for securing IoT-Fog environments against Slow HTTP-DoS attacks whilst preserving the integrity of attack evidence [10], [11]. It provides a solid foundation for more advanced future work, notably full automation, real-world distributed deployment and improvement of the detection model.

References

[1] H. F. Atlam, A. Alassafi, A. Alenezi, R. J. Walters, et G. B. Wills, "Identity Management and Authentication in the Internet of Things," *Healthcare Informatics Research*, 2018.

[2] F. Bonomi, R. Milito, J. Zhu, et S. Addepalli, "Fog Computing and its role in the Internet of Things," *ACM (MCC Workshop)*, 2012.

[3] I. Sharafaldin, A. H. Lashkari, et A. A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization," *ICISSP*, 2018.

[4] A. L. Buczak et E. Guven, "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection," *IEEE Communications Surveys & Tutorials*, 2016.

[5] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," *Ethereum Project Yellow Paper*, 2014.

[6] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," *Cryptography Mailing List*, 2008.

[7] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, et R. Ranjan, "IOTSim: A simulator for analysing IoT applications," *Journal of Systems Architecture*, 2017.

[8] M. A. Ambusaidi, X. He, P. Nanda, et Z. Tan, "Building an Intrusion Detection System Using a Filter-Based Feature Selection Algorithm," *IEEE Transactions on Computers*, 2016.

[9] P. Zhang, M. Zhou, et G. Fortino, "Security and trust issues in Fog computing: A survey," *Future Generation Computer Systems*, 2018.

[10] R. Khan et A. S. Namin, "Blockchain-Based Mitigation Framework for Distributed Denial of Service Attacks in IoT Networks," *IEEE Access / Int. Conf. on Blockchain*, 2022.

[11] A. A. Diro et N. Chilamkurti, "Distributed attack detection in IoT using Deep Learning approach at the Fog layer," *IEEE Transactions / Future Gen. Comp. Syst.*, 2022.

[12] G. Maciá-Fernández, J. E. Díaz-Verdejo, et P. García-Teodoro, "Evaluation of a low-rate DoS attack against iterative servers," *Computer Networks*, 2007. *(Référence pour les attaques Slow-HTTP-DoS)*

[13] Ethereum Foundation, "Documentation officielle de Solidity (version 0.8.x)," En ligne : https://docs.soliditylang.org.

[14] Z. Zheng, S. Xie, H. Dai, X. Chen, et H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," *IEEE International Congress on Big Data*, 2017.

[15] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, 2011. *(Référence pour l'utilisation des scalers et modèles .pkl)*