# Leiden University

## ICT in Business and the Public Sector

A privacy-preserving fault detection system for IoT networks using blockchain, MPC, and edge computing


| | |
|---|---|
| Name: | Lars Nieuwdorp |
| Student ID: | s4048865 |
| Date: | 11-8-2025 |
| 1st supervisor: | Dr. Ir. Eleftheria Makri |
| 2nd supervisor: | Prof. Dr. Ir. Nele Mentens |


MASTER'S THESIS


Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Einsteinweg 55
2333 CC Leiden
The Netherlands

# Abstract

**Background**
The use of IoT devices in security-sensitive environments enables monitoring but usually relies on centralized fault detection. This creates a single point of failure and risks exposing sensitive data when multiple branches wish to share security logs. An approach that combines local edge computation, blockchain with a hybrid on- / off-chain logging system for tamper-evident yet erasable records and multi-party computation (MPC) for secure risk aggregation can address these issues while meeting privacy and audit requirements.

**Aim**
Research, at a system level, the integration of edge processing, blockchain, hybrid on- / off-chain logging, and MPC for fault detection, and deliver a modular proof-of-concept framework that demonstrates feasibility and highlights concrete optimization areas for future work.

**Method**
We propose a framework where each branch conducts local fault processing and storage on the network edge, encrypts and logs events off-chain, stores hashes on a private Ethereum network, and uses MPC to calculate a global threat score (a combined threat score). We develop a fully containerized working prototype and carry out controlled experiments with different loads to assess latency, resource utilization, network traffic, and power consumption.

**Results**
In our most demanding experimental setup, in which we ingested 2,000 synthetic fault events over a three-hour window and executed eight scheduled MPC computations, end-to-end latency remained below 14 ms. Meanwhile, CPU time reaches approximately 400 CPU seconds, memory usage peaks at 3.3 GB, and the average power consumption increases to 8.6 watts above the measured baseline. MPC adds an additional 2.3 seconds per party for each processing trigger and requires 240 MB of communication for each party involved. For the other scenarios, these values vary slightly.

**Conclusion**
In our test environment with the twelve test scenarios, the prototype demonstrates that end-to-end feasibility is achievable while offering data privacy and tamper-evident logging. The framework sets up baselines and interfaces that can be improved upon in future projects.

# Acknowledgements

I thank Dr. Ir. Eleftheria Makri for her guidance throughout this thesis; her feedback refined my research and analysis. I also thank Prof. Dr. Ir. Nele Mentens for her careful evaluation of this thesis.

# Contents

# Chapter 1

# Introduction

## 1.1 Thesis structure

- **Chapter 1** introduces the research problem, states the aim, and lists the main contributions.
- **Chapter 2** defines key concepts and terminology.
- **Chapter 3** describes the methodology used in the thesis.
- **Chapter 4** reviews the literature that informs the design choices made in this thesis.
- **Chapter 5** presents the system architecture and the main design choices.
- **Chapter 6** describes how the proof-of-concept prototype is built and deployed.
- **Chapter 7** presents the performance metrics and privacy outcomes.
- **Chapter 8** interprets the results, answers the research questions, and discusses limitations.
- **Chapter 9** summarizes the conclusions and outlines directions for future work.

## 1.2 Introduction

This thesis researches and presents the design and implementation of a proof-of-concept privacy-preserving system for fault detection in distributed IoT networks, with a focus on sensitive locations like banking facilities.

There are many types of IoT devices in use across diverse sectors, also in critical infrastructure areas. These devices help facilitate operations and can, for example, help to comply with security standards. This means that malfunctions or security failures in IoT monitoring systems can present risks.

The proposed privacy-preserving fault detection and storage framework aims to help security-sensitive environments to exchange security-relevant information to identify coordinated attacks across branches securely and enables IoT devices to log faults while enabling tamper-evident auditability for these logs through the use of blockchain. The system allows for the deletion of the log contents when required, through the use of a hybrid on-chain / off-chain storage system.

The system runs on the network edge and identifies / processes received faults from IoT devices. The private blockchain-based hybrid chain storage system, also running on the network edge, will log this data securely, ensuring tamper-evident records that help with both auditability and regulatory compliance.

A secure multi-party computation (MPC) component that is part of the system ensures that a global threat detection process can happen securely between branches without exposing any raw security data. Each branch generates data when a fault is detected and, collectively with other branches, calculates a threat score that allows for proactive tightening of security at other branches, even if no faults have been found at a different branch (yet). This can help identify coordinated attack patterns across branches before they escalate into widespread security incidents.

This system aims to enable secure, tamper-evident, real-time fault insights across branches while safeguarding sensitive information.

## 1.3 Research problem

The use of IoT devices like cameras and tag readers in security-sensitive environments, such as bank vaults, can help improve security monitoring. These devices can help facilities secure their assets through real-time monitoring to prevent unauthorized access and ensure safety standards [1, 2]. The implementation of these devices, however, can also introduce security and privacy challenges [2]. Failures or malfunctions in these devices may result in risks such as unauthorized access, potential data breaches, and non-compliance with regulations [2]. Monitoring these devices for failures is therefore important.

Centralized systems transmit and store sensitive fault data on a central server, creating a single point of failure that can expose device information [3, 4].

Blockchain-based IoT logging systems can improve availability (due to the distributed architecture of blockchain) and auditability (cryptographically guaranteeing that data cannot be tampered with) by storing this data on-chain [4]. However, storing detailed event data on the ledger can conflict with privacy requirements and regulations, for example, the right to erasure that are part of regulations like the GDPR [5], especially if the data contains information about customers [6, 5].

Also, in a multi-branch environment, security-sensitive environments may miss coordinated attacks unless they share data between locations, but sharing raw data can expose local vulnerabilities to other branches. For example, in a bank setting, this can expose security-specific vulnerabilities to another branch. A privacy-preserving MPC system would let these environments share data without revealing sensitive details [7]. This method can help prevent coordinated attacks and as a result reduce downtime in high-security settings.

MPC and blockchain-based systems, however, are often unsuitable for resource-constrained environments [8, 9], since these technologies are often computationally expensive. Edge computing can help by offloading these computational tasks while being close to IoT devices, keeping the data at the branch-level [8, 9]. This reduces latency and bandwidth consumption, enables real-time processing and preserves data privacy [10, 8].

To our knowledge, no prior framework integrates MPC, blockchain and edge computing into a single, branch-level system that delivers low-latency fault detection, tamper-evident audit trails and privacy compliance. This research aims to develop such a system, which has the potential to enhance data privacy while increasing security insights, and ensure compliance with the increasingly stricter regulatory and security requirements, such as in the banking sector [11].

## 1.4 Research questions

This thesis focuses on the following main question:
*How can an efficient fault detection system be designed for high-security IoT applications while ensuring real-time, privacy-preserving, and tamper-evident security monitoring?*

This question is split into the following sub-questions:

**SQ1**: How does a decentralized fault detection system leveraging blockchain improve the security and integrity of fault logs compared to a centralized alternative?

**SQ2**: What performance penalty does multi-party computation introduce in high-security IoT fault detection?

**SQ3**: What are the trade-offs between decentralization, auditability, and regulatory compliance in an IoT-based security monitoring system?

**SQ4**: How do latency, throughput, and resource overhead compare between system configurations with and without blockchain and MPC?

## 1.5 Scope

A system-level feasibility study of integrating edge processing, blockchain with hybrid on- / off-chain logging, and MPC at branch level. We deliver a working, modular prototype with replaceable components and report baseline measurements from controlled, three-branch lab experiments.

## 1.6 Contribution

This thesis makes contributions to the design, implementation and evaluation of privacy-preserving fault detection in high-security IoT networks. It builds upon the knowledge and gaps identified in the literature review (Chapter 4).

1. We propose a high-level, modular framework that combines edge computing, MPC and a private blockchain with hybrid on- / off-chain logging to detect, correlate and log IoT device faults without exposing raw security data.
2. We develop and open-source a containerized prototype, including an IoT simulator (simulating IoT devices sending faults), a Flask-based coordinator (the main logic running at every branch), MPC parties and a proof-of-authority based private Ethereum network that implements the proposed architecture end-to-end.
3. We implement a hybrid on-chain / off-chain storage mechanism, as first proposed in the literature, that stores only cryptographic hashes on-chain (ensuring immutable audit trails) while encrypted full logs remain off-chain and can be deleted or changed to help with GDPR requirements. This specific use case is a gap identified in the implementation of blockchain in such environments.
4. We design and implement a weighted-average MPC protocol in MP-SPDZ that computes a global "threat score" (a percentage) from per-branch severity sums and counts entirely within the secret-shared domain (preserving branch-level confidentiality).
5. We perform a series of controlled component-ablation experiments (by disabling the blockchain and MPC system in the prototype) to compare the system against a baseline. We measure CPU and memory usage, disk storage utilization, power draw, network traffic (communication costs) and latency within the system. This allows us to measure the impact of each component.

These contributions aim to advance both the theoretical understanding and the practical engineering of secure, privacy-preserving fault detection systems in high-security IoT environments.

# Chapter 2

# Background

## 2.1 Terminology and concepts

This section outlines the fundamental concepts and technologies that form the basis for designing and implementing the privacy-preserving, decentralized fault detection system.

### 2.1.1 Internet of things

The internet of things refers to a network of interconnected devices, such as sensors, cameras, actuators, and embedded systems, that collect and exchange data over communication protocols with or without human intervention [12].

### 2.1.2 Blockchain

A distributed ledger that keeps track of transactions through a series of interconnected blocks, where every block contains a collection of records along with a cryptographic connection to the previous block [4]. Participating nodes hold the chain of blocks [4]. After data is added to a block and the block has been signed (sealed) through the network's consensus mechanism, it becomes immutable; any attempt to alter that data would compromise the cryptographic links [4]. We use a private, permissioned blockchain based on the proof-of-authority consensus mechanism to securely record fault hashes and global threat scores in a manner that is resistant to tampering.

### 2.1.3 Consensus mechanism

The method through which blockchain nodes reach a consensus on the next block to be added. In proof-of-authority (PoA) specifically for example, a predetermined group of trusted nodes are allowed to "seal" blocks (validate transactions) [13]. Consensus ensures that all honest nodes have access to the same immutable ledger [13]. These are fundamental to maintaining the integrity and security of blockchain systems.

### 2.1.4 Hybrid on-chain / off-chain storage

A pattern that was initially suggested in earlier research [6] and aims to distinguish large or mutable data from the blockchain itself. A cryptographic hash of each event log is recorded on-chain, while the complete encrypted log remains stored off-chain in local storage [6]. This approach allows for the deletion or modification of information to ensure compliance with GDPR [5] and/or internal guidelines, all while maintaining an unchangeable record of their original contents (a computed hash) on-chain.

### 2.1.5 Multi-party computation

Multi-party computation is a set of protocols that enables several participants, referred to as parties (in this thesis, each branch), to collaboratively compute a collective function, like the average fault severity between all branches, based on their private inputs (in the case of this thesis, their local sums and

counts) while keeping those inputs confidential from one another [7]. MPC protocols rely on building blocks such as secret sharing, where inputs are split into shares and distributed between the parties, and then computed on without revealing private data [7, 14]. In the end, they only collect the final result, which is the collective outcome, rather than the raw data from each individual branch [7].

### 2.1.6 Secret sharing

Secret sharing involves dividing a confidential value $s$ into multiple random components, known as "shares", ensuring that no single individual can independently discover the original number $s$ [14]. Through the use of secret sharing, it becomes possible to carry out computations on private data, such as sums or averages, while ensuring that no participant's value is revealed. Only the final outcome is revealed.

Secret-sharing schemes are generally instantiated over an algebraic domain $A$ (for example, a finite field $\mathbb{F}_p$ or a ring such as $\mathbb{Z}/2^k\mathbb{Z}$), with the following requirements [15]:

- You can draw shares uniformly (or pseudorandomly) from $A$, so that each individual share leaks no information about the secret.
- All additions, multiplications, and other necessary operations are performed inside $A$.
- There is a well-defined combining procedure that recovers the secret whenever the threshold of shares is met.

There are various methods for carrying out secret sharing. We explain the concept of secret sharing using additive secret-sharing, as it provides a straightforward approach.

#### Additive sharing (all-of-$n$)

In this secret sharing method, every one of the $n$ parties must combine their share to recover $s$. For $n = 3$, it works as follows:

1. Pick a prime $p$ bigger than the secret $s$.
2. Choose two random numbers $r_1, r_2$ in $\{0, \ldots, p-1\}$.
3. Let the third share be
$$r_3 = \big(s - (r_1 + r_2)\big) \bmod p.$$
4. Give party 1 the number $r_1$, party 2 the number $r_2$, and party 3 the number $r_3$.

Each share on its own is random (because it is a random number mod $p$). Only when we add all three (in the same modulo), we get back

$$(r_1 + r_2 + r_3) \bmod p \;=\; s.$$

*Example.* When we share $s = 42$ with prime $p = 101$, we:

1. Pick $r_1 = 10$, and $r_2 = 25$.
2. Compute $r_3 = (42 - (10 + 25)) \bmod 101 = 7$.
3. Shares are $(10, 25, 7)$.
4. Check: $(10 + 25 + 7) \bmod 101 = 42$.

*Example 2.* Modular arithmetic wraps negative results back into the field range $\{0, \ldots, p-1\}$. This means that when we share $s = 3$ with prime $p = 7$, we:

1. Pick $r_1 = 5$ and $r_2 = 6$.
2. Compute
$$r_3 \;=\; \big(3 - (5 + 6)\big) \bmod 7 \;=\; (-8) \bmod 7 \;=\; 6.$$
3. Shares are $(5, 6, 6)$.
4. Check:
$$(5 + 6 + 6) \bmod 7 = 17 \bmod 7 = 3.$$

#### Other methods of secret sharing

In addition to additive sharing, there are various different techniques that are can be used. One example is Shamir's $t$-out-of-$n$ threshold sharing [16], where any $t$ out of $n$ parties can recover $s$, but any single party can still learn nothing on their own.

There are many other methods of performing secret sharing, which can all differ in the access structures they support, the efficiency of share generation and reconstruction, the size of the shares, and the assumptions they rely on, such as computational versus information-theoretic security [16].

#### Secret sharing in SPDZ

SPDZ protocols [17, 18, 19] rely on the same foundational concepts of secret-sharing, but can also incorporate authentication codes (MACs) and preprocessing techniques (Beaver triples) to facilitate secure multi-party processes.

#### Beaver triples

Beaver triples are random tuples $(a, b, c)$ that are secret-shared among parties and are characterized by the relationship $c = a \cdot b$. These tuples are generated during an offline preprocessing phase, before any inputs are known. Shifting the expensive task of creating these multiplication helpers to the offline phase allows the online phase (when we actually multiply the secret-shared values $x$ and $y$) to require fewer local additions and fewer rounds of communication [18, 20]. In other words, this lets participants bypass heavy work during the online phase.

This process combines shares of $x - a$, $y - b$, and $c$, eventually producing shares of $x \cdot y$ without ever disclosing $x$ or $y$ [18, 20]. This separation between offline and online processes ensures the computationally expensive work is completed beforehand, allowing the online phase to be quick while ensuring that all contributions remain confidential.

#### Message authentication codes (MACs)

MACs are a verification process for each secret share to ensure that it remains untampered with [18]. In SPDZ$^{2k}$ [19] for example, every party holds additive shares of a single global secret MAC key. Each value that is secret-shared is accompanied by a hidden authentication tag computed as "key times value". The tags are shared in a secretive manner, ensuring that no one is able to discover the key or any complete tag.

When a value is opened, only that specific value is made public, while the tag shares remain concealed. The parties continue to conduct a standard batched check, a MACCheck. This process mixes multiple opened values with new randomness and ensures that all their hidden tags align consistently with the same secret key, all while keeping the key itself confidential.

If this check does not pass, it indicates that at least one share has been modified [18]. This allows parties to detect cheating while keeping the secret values confidential, and the parties abort.

### 2.1.7 MPC frameworks and SPDZ$^{2k}$

SPDZ$^{2k}$ [19] is a secure MPC protocol that operates over a ring of size $2^k$. It means that all values shared in secret and the arithmetic operations (such as additions, multiplications, and fixed-point encodings) are conducted modulo $2^k$. Also, each share includes a MAC, which ensures that any tampering can be identified and active security guarantees can be provided.

In high-security environments, it can be important to select an MPC protocol that ensures the confidentiality of inputs while also being resilient against active attackers. SPDZ$^{2k}$ provides security in scenarios where there are dishonest majorities, which are situations where over half of the participants might work together in a harmful way. SPDZ$^{2k}$ maintains security in this model through the use of an offline preprocessing phase, using a scheme that uses authenticated sharing and message authentication codes [19]. We distinguish:

- Active vs. Passive Security

  - Passive ("honest-but-curious") protocols ensure privacy protection only when every participant adheres to the established protocol specifications, yet they may attempt to extract additional information from the messages they observe. [14].
  - Active-secure protocols ensure that correct results are achieved, even when corrupted parties may deviate arbitrarily from the protocol, such as by sending improperly formatted messages [14].
- Offline preprocessing
  - MPC protocols can divide the secure computation process into two phases: an offline preprocessing phase, which is slower, and a quicker online phase. For example, by precomputing Beaver triples, the online phase can be streamlined [18, 20].
- Performance
  - Protocols can vary in terms of their computational costs, such as CPU cycles and memory usage, as well as their communication volume, which includes factors like round-trips and total bytes transmitted.

### 2.1.8 Edge computing

A distributed computing implementation that reduces latency and bandwidth consumption by processing data nearer to the source. For example, by placing a server directly in a branch instead of the cloud [10]. Edge computing enables real-time processing in IoT environments [8], enhances data privacy by keeping sensitive data local [10], improves energy efficiency [9] and increases data security [21].

### 2.1.9 Encryption and verification methods

In the proposed system, it is important that we protect detailed fault logs, ensuring that only authorized individuals have access to them. This means we need a method to verify that these logs remain secure. We use two techniques for this purpose: the first is AES-CBC, which secures (encrypts) the log contents, and the second is SHA-256, which generates a fingerprint (hash) of the fault data.

#### AES-CBC

To keep full logs that are kept off-chain private, we encrypt each log entry before storing it. We use AES in Cipher Block Chaining (CBC) mode, which works like this:

- Example: A branch's camera sends a new fault report (for example, "camera blurred at vault"). Before storing the report, we process it with the AES-CBC algorithm so that only authorized individuals can view the contents.
- **Simplified, this algorithm works as follows:**
  1. We divide the log text into chunks of equal size (blocks).
  2. We combine each chunk with the previously scrambled chunk, using a random starter value for the first chunk.
  3. Next, we use a secret key to convert the mixed chunk into ciphertext.
- The encrypted text is unreadable without the secret key [22, 23].

#### SHA-256

We process the fault data using SHA-256 to generate a fingerprint (a hash), which is a fixed-length 256-bit digest, using the SHA-256 algorithm. This process is a mathematical mixing technique that consistently generates a 256-bit output. A slight alteration to the input results in an entirely different hash. SHA-256 is used to verify that files or communications remain unchanged; if the hashes match, the contents have not changed [24].

### 2.1.10 Hyperledger Besu

Hyperledger Besu is an open-source Ethereum client. The system implements the Ethereum Virtual Machine (EVM) and allows for private, permissioned networks. We use Clique (EIP-225) mode [25] for the consensus mechanism. Besu provides a JSON-RPC interface (which includes the web3 API) along with detailed account and node permissioning [26]. This helps easy integration with Web3.py in microservice deployments, and with integration into software systems.

# Chapter 3

# Method

This research uses a design science methodology [27]. We have a top-level goal (solving a practical problem) and focus on the creation and investigation of an IT artifact (the creation of the privacy-preserving fault detection framework). We validate the designed artifact using lab experiments and benchmarking.

## 3.1 Literature review

The methodology for the literature review in the thesis involves a search and selection of relevant studies, filtered on the inclusion criteria.

### 3.1.1 Inclusion criteria

The following criteria were applied to select relevant studies:

- We primarily focus on publications from the past decade to maintain relevance.
- Studies focusing on relevant information on blockchain, MPC, or edge computing in IoT (or IoT fault detection).
- Papers addressing IoT security challenges in high-security environments.

### 3.1.2 Related work

The literature discussed in Chapter 4 established the foundation for the system design, influencing important architectural choices and guiding the approach to experimental evaluation. This research was influenced by related work, including but not limited to:

- Research regarding the integration of blockchain in IoT fault detection guided the choice to implement a blockchain system that uses a lightweight consensus protocol. The use of Proof-of-Authority, as discussed by Azbeg et al. [13], enables the use of blockchain while avoiding high computational demands, such as the ones linked to Proof-of-Work [9].
- Studies and information on MPC, also in settings where privacy is important (including the use of MPC in environments with limited resources), including theories behind and practical use of MPC [7, 8].
- Interpreting the findings from research conducted by Zalte-Gaikwad [10] and Liu et al. [8], the role of edge computing is important in minimizing data transfer and facilitating real-time responses in fault detection systems. The research papers also highlight the importance of integrating the logic near the data source, helping with low latency and real-time processing.
- In research papers by Li et al. [6] and Uddin et al. [3], GDPR [5] and details on hybrid storage models influenced the choice to implement a hybrid on-chain / off-chain storage model, which stores full encrypted logs off-chain. This model could for instance solve the conflict between immutability and data erasure rights, a challenge that has been identified.

## 3.2 System design

### 3.2.1 Design science phases

The design science methodology approach used is organized into three iterative phases. The phases depend on insights from prior studies detailed in the literature review and aim to address security and privacy issues related to IoT fault detection in high-security settings. The phases are:

1. **System design/architecture**: The initial phase focuses on defining the overall system architecture, detailing the components, data flows, and security mechanisms involved.
2. **Prototype development**: A prototype is developed and deployed using containerized services, following the architecture designed in the previous phase.
3. **Evaluation and comparative analysis**: The final phase evaluates the system. The evaluation is based on data gathered from controlled tests and is compared to the system without MPC and Blockchain. The prototype is used to evaluate the feasibility and performance of the system.

### 3.2.2 Prototype and simulation setup

A functional prototype of the proposed system is designed and evaluated under simulated conditions to evaluate its performance. Every branch operates (1) an IoT fault generator, (2) a coordinator service for local orchestration, processing and storage, (3) a Hyperledger Besu node in Clique mode for on-chain storage and verification for off-chain logs, and (4) an MPC party utilizing MP-SPDZ (SPDZ$^{2k}$). Fault logs are encrypted using AES-CBC and are stored off-chain, while only SHA-256 digests and aggregate threat scores are stored on-chain.

# Chapter 4

# Literature review

## 4.1 Summary

The research in privacy-preserving computing explores different approaches to address the growing scale and security concerns associated with IoT networks. High-security environments such as financial institutions, industrial control systems, and government infrastructure that uses IoT may need effective fault detection mechanisms to ensure protection of privacy. Three primary technological approaches have been investigated to meet these needs; MPC, edge computing, and blockchain integration. Each of these technologies has its own distinct benefits and limitations.

First, MPC represents an important advancement in the field of privacy-preserving computation because it allows several participants to work together while ensuring that their private inputs remain secure. For example, this can be a collective weighted severity score between participants. This is useful in scenarios where various parties need to analyze this shared data while also ensuring confidentiality is preserved.

MPC proves to be effective for this secure collaboration; however, it may struggle with the resource limitations associated with IoT devices. Edge computing therefore can be used as an additional strategy by bringing computation closer to the data source, without relying on the devices themselves or centralized cloud infrastructure. This enables the execution of more computationally intensive programs at the edge by offloading them to these edge devices. This helps prevent the IoT devices from becoming overloaded while also not interfering with their essential functions.

Edge computing provides advantages, for example by reducing reliance on centralized cloud servers, but it can also introduce challenges. One challenge identified is data integrity and trust. In the absence of a central authority, there exists no "good" method to make sure that the results of computations remain unchanged. Blockchain technology offers a practical approach to this problem by creating a decentralized verification system that ensures data remains unchanged and can be traced back.

Blockchain networks need a validation mechanism, known as a consensus mechanism. The literature has investigated a range of consensus mechanisms, such as proof-of-authority, proof-of-stake, and practical byzantine fault tolerance, all of which aim to minimize energy consumption and computational demands. These protocols help nodes come to a consensus while maintaining important safety principles. These include data integrity, which prevents tampering; network availability, which ensures that processes continue even in the presence of failures; and fault tolerance, which offers resilience against both malicious attacks and malfunctioning nodes.

One of the main obstacles that blockchain-based systems encounter, especially in high-security applications, is adhering to the strict regulations that data privacy. This suggests that there exists a trade-off between privacy and the need for auditability. Blockchain alone may not comply with legal standards and guidelines, because the immutability of blockchain ensures data integrity, but it also makes it challenging to modify or delete information. This characteristic is particularly noteworthy when it comes to compliance with regulations such as the General Data Protection Regulation (GDPR). hybrid on-chain / off-chain storage mechanisms could offer a solution to this problem.

The literature also examines common IoT security challenges, including device vulnerabilities, centralized

failure risks, and compliance requirements in high-security contexts.

## 4.2 Thematic review

We split the literature review into areas, with each area focusing on the most relevant aspects of the proposed research; blockchain technology, MPC, and edge computing. In each area, we highlight the findings and limitations.

### 4.2.1 Theme 1: Blockchain combined with IoT fault detection

#### Summary

Blockchain uses cryptographic techniques along with a decentralized framework. It can guarantee data integrity and create a secure audit trail that is resistant to manipulation. Real-world implementation in IoT environments can be restricted by factors such as resource limitations and regulatory compliance. Adjusting the system to incorporate optimized consensus mechanisms and offloading can make it capable of addressing these limitations in resource-constrained environments. This can enhance the viability of blockchain for real-world IoT fault detection applications.

#### Literature review

When overcoming the limitations IoT environments typically have, blockchain technology can be a promising solution to help address challenges in IoT devices (as discussed in Chapter 4.2.5), and specifically can help facilitate fault detection for these devices.

A decentralized architecture can address the vulnerabilities associated with centralized systems. In particular, it can address the single points of failure that might compromise the overall functioning of the system. This is related to the distributed nature of the system, where nodes validate transactions and store redundant record of the ledger (each node stores one copy of the ledger) [3, 4]. This prevents a node failure from affecting overall system integrity and function [3, 4].

Blockchain uses cryptographic techniques to guarantee that all transactions and identified faults, such as logs and other data, are recorded in a verifiable, tamper resistant ledger. When set up correctly, this creates an immutable audit trail [4, 28, 6]. This feature can become particularly important in situations that demand accountability and trust, such as when it is necessary to maintain a record of immutable logs.

Efficiency in both computing power and energy consumption has been an important concern in the adoption of blockchain technology within constrained environments, as highlighted by studies discussed in Chapter 4.2.2. This emphasis particularly is related to consensus mechanisms, which are protocols employed in blockchain networks to ensure that distributed nodes reach an agreement regarding the validity of transactions and the current state of the ledger [13]. Consensus mechanisms guarantee that all participants (nodes) maintain a consistent and tamper-evident record of data, without depending on a central authority [13, 4]. These aspects help maintain the integrity and security of blockchain systems. It is therefore important to select an appropriate consensus mechanism. Chapter 4.2.2 investigates the comparison of consensus mechanisms that are most appropriate for IoT environments.

The integration of blockchain and edge computing in IoT networks offers a promising way to enhance both efficiency and security by combining their respective strengths. More powerful edge devices can take on the blockchain operations (which generally require heavy computational power) for IoT nodes. Running transaction validation, block creation, and consensus protocols at the network edge allows resource-constrained IoT devices, which may not be able to handle these tasks, to offload these tasks [29, 9]. This makes blockchain a more feasible option for IoT applications. The research into utilizing edge-computing-assisted mechanisms to optimize blockchain's performance in IoT environments is explored further in Chapter 4.2.4.

Blockchain's practical applications in IoT, specifically in the field of log storage, have been explored in the studies, such as the study by Li et al. [6]. Here, blockchain facilitates tamper-evident storage and the system allows for efficient querying of log data, which can be done by integrating on-chain and off-chain storage solutions [6]. This storage approach is also explored in other studies, such as the study by Uddin et al. [3]. This storage design could also enable users to maintain a verifiable and tamper-evident record

on the blockchain, while ensuring that private data is stored off-chain, which would enable the ability to delete or modify off-chain data upon request.

However, implementing blockchain technology in IoT settings highlights several challenges. The first issue is the overhead associated with blockchain. IoT devices are typically characterized by their limited memory and computational resources [3]. This makes it difficult to accommodate blockchain's resource requirements [30, 3].

As a result, one effective solution can be to offload data to edge nodes. Edge nodes, which can be equipped with better resources, can act as intermediaries in this model [9]. When we assign the blockchain process to these nodes, IoT devices are able to avoid the heavy resource demands that come with maintaining a complete blockchain ledger [9, 3, 30]. This could expand the application of such a system.

Blockchain guarantees that fault detection logs (stored as transactions in the blockchain system) are immutable and verifiable across nodes, which improves data integrity, tamper resistance, and trust in distributed scenarios. But, consensus algorithms such as Proof-of-Work can result in increased computational cost, power consumption and latency, therefore its transaction processing speed may not satisfy real-time limitations in IoT scenarios [31, 32]. This means that while blockchain enhances security, its transaction processing speed, at least for the aforementioned consensus mechanism may not meet real-time requirements in fault detection scenarios, unless specifically addressed. The added latency and scalability can present additional challenges [13, 31]. Other consensus algorithms may be necessary to address these performance bottlenecks. This is further researched and tested in Chapter 4.2.2.

Regulatory and compliance challenges introduce additional complexity, as the decentralized and immutable characteristics of blockchain may conflict with data privacy laws like the GDPR[5]. This is particularly relevant in situations where blockchain is used for storing sensitive information, such as personally identifiable details regarding customers or employees. In such a situation, maintaining compliance while preserving the fundamental advantages of blockchain can become an issue [3, 31]. hybrid on-chain / off-chain storage mechanisms could offer a solution.

Li et al. [6] explore a hybrid on-chain/off-chain log architecture that focuses on scalable, tamper-evident storage and efficient search. In their model, they maintain complete (encrypted) log payloads off-chain, while committing only minimal on-chain data (cryptographic pointers (e.g., hashes), keyword indices, and timestamps) to ensure integrity and auditability. Their work presents an opportunity for adaptation to our specific use case, although the paper does not provide a modification/deletion mechanism.

#### Findings and conclusions

Our analysis shows that blockchain could improve the security of IoT fault detection by using a decentralized and tamper-resistant ledger. It uses cryptographic hashes alongside a distributed node architecture, which removes single points of failure and ensures the integrity of data. And by transferring validation and storage tasks to more capable edge nodes we can allow constrained IoT devices to avoid heavy computation and the ledger maintenance tasks. Hybrid storage designs that combine on-chain and off-chain elements could provide immutable audit trails while possibly also allowing modifications or deletions of sensitive data off-chain to meet, for example, regulatory requirements, but this aspect needs further research. Consensus mechanisms that are suitable for resource-constrained environments along with edge-assisted frameworks need to be researched in order to effectively support real-world IoT deployments.

### 4.2.2 Theme 2: Consensus mechanisms for IoT suitability

#### Summary

Proof of Work secures data by relying on intensive computational processes; however, it consumes excessive energy. Proof of Stake reduces energy consumption; however, it also creates risks such as centralization, "nothing-at-stake" issues. PBFT can handle faulty nodes; however, it can experience slowdowns when faced with real-world workloads. Proof of Authority uses a limited and trusted set of validators to achieve low latency and minimize energy consumption.

### Literature Review

Consensus mechanisms are the foundation of blockchain networks. They enable agreement between distributed nodes while ensuring the integrity and security of the ledger [13, 28]. In IoT-based fault detection specifically, consensus enables nodes to collectively verify and confirm transactions that contain fault data (which are data elements, such as strings) independently of a central authority. Ensuring trust in the data stored within this ledger is important, as it helps to prevent any potential manipulation of that data.

In blockchain systems tailored for IoT environments, the choice of a suitable consensus mechanism can depend on various factors, including security, scalability, energy efficiency, and the limitations that come with resource-constrained devices. This analysis examines different consensus mechanisms and evaluates how suitable they are for IoT systems, particularly in high-security environments like bank vaults. The application of blockchain consensus mechanisms in IoT contexts has been a focus of research, since each mechanism offers distinct advantages and faces unique limitations when applied to IoT environments [13, 28].

### Comparisons between consensus mechanisms

Proof-of-work (or PoW) is the technology behind the Bitcoin network, Litecoin, and others [13]. PoW operates through "miners" solving complex cryptographic puzzles that demand considerable computational resources. This is by design, the protocol is intentionally structured to ensure that the process is computationally expensive and time-intensive [33, 28]. This is what makes the network secure; the security of the network is tied to its computational power, which ensures data immutability on the blockchain. This makes it difficult for an attacker to modify past transaction blocks, as they would need to redo the work for that specific block and all subsequent blocks, while also needing to catch up to and surpass the cumulative work of the chain [33].

The disadvantage of this design is its biggest inefficiency; substantial energy is consumed to power the mining hardware (the computers that solve the cryptographic puzzles) [13, 28]. The Proof of Work mechanism has faced criticism due to its environmental impact, as the electricity consumption often compares to that of countries. For instance, the energy consumption of the Bitcoin network matches or surpasses that of certain countries, including Denmark, Chile, Finland, and The Netherlands [34]. This indicates that conventional consensus mechanisms, such as Proof of Work, tend to be resource-heavy and therefore are probably not ideal for IoT environments.

PoW is not the only blockchain consensus mechanism protocol, there are other alternatives such as Proof-of-Stake (PoS), Proof-of-Authority (PoA), among others. These mechanisms achieve consensus while reducing the energy demands that are associated with PoW [13]. In PoS, for example, validators are chosen to create new blocks and confirm transactions based on the process that takes into account the validators' stake. This avoids the need for energy-intensive computations as required in PoW [13]. To demonstrate the energy and compute savings, the ethereum network is a good case study. Ethereum's transition from PoW to PoS in September 2022 (known as "the merge") reduced its energy consumption by approximately 99.9 percent[35, 36].

The approach used in Proof of Stake, known for its scalability and energy efficiency [13], appears to be quite promising. However, it also can bring up concerns regarding centralization and potential issues, including the "nothing-at-stake" problem [28, 13]. In Proof of Stake, validators are selected to create and confirm new blocks depending on the coin age they have staked [13, 37]. In contrast to Proof of Work, which demands notable computational resources, Proof of Stake allows validators to lower energy consumption [37].

This approach can also introduce important issues. First of all, Proof-of-Stake can suffer from the "nothing-at-stake" problem. Because validators can build on every competing fork at no real cost, they can undermine transaction finality and enable double-spend exploits [37, 13]. Also, Proof-of-stake concentrates power with large stakeholders, who can outvote smaller holders to centralize block production, censor transactions, or influence protocol changes in their advantage [13, 3]. At last, PoS lowers the barrier for long-range history attacks (which means that it lets attackers with enough coin-age alter old blocks) and force changes in the blockchain [37]. This means that when users cannot rely on the integrity of transactions (the fairness of blocks and the "permanence" of history), their confidence in decentralization decreases.

Another consensus mechanism, Practical Byzantine Fault Tolerance (PBFT in short) works by requiring nodes in the network to agree on the system's state, even if some nodes act maliciously or fail [38]. PBFT functions through a three-phase protocol consisting of Pre-Prepare, Prepare, and Commit. This process is managed by a primary node that assigns sequence numbers to client requests, ensuring a total order is established before execution and response [38]. PBFT can handle up to one-third of the nodes being faulty or malicious while still ensuring the system remains operational and secure [38].

However, there are practical issues when running PBFT in production. First, PBFT requires that the primary node (the designated leader) deterministically order and broadcast client requests, ensuring that every backup replica executes the same state-machine transition using identical inputs and in the same sequence, thus arriving at the same state. Each replica is also responsible for validating these inputs, which functions well in a steady state; however, time drift can disrupt validation, causing the entire system to stall [39]. Additionally, since PBFT is leader-based; it is also less decentralized than leaderless peer-to-peer designs.

Also, the true cost of disk usage and cryptographic checks are overlooked by benchmarks that display tens of thousands of null operations per second. When you transition this system to use "real data", the throughput decreases by up to 100 times in comparison to those idealized figures [39]. End users could possibly experience slow transactions (performance degradation) or timeouts.

Proof of Authority, or PoA, relies on a limited number of trusted, pre-approved validators. In this model, the system relies on identity and reputation, meaning that the nodes involved in the network are motivated to act appropriately in order to maintain their reputation [13, 28]. Validators are chosen prior to deployment and are trusted, which eliminates the necessity for resource-intensive processes to reach consensus [13, 28]. Because only a small, trusted set of nodes participates, PoA avoids energy-intensive mining and limits network traffic. This results in higher throughput and better performance compared to PBFT systems [28], while also being energy-efficient [13].

However, the PoA model also has unique challenges. First, fair processes are important for selecting validators, as they can help maintain trust and prevent a centralization of power. Choosing validators is important for the trust model (since it is authority based), also making it essential for this process to be transparent to avoid reducing the system's overall trustworthiness [13, 28]. One system that has been proposed to solve this in the Ethereum Clique PoA specification is an on-chain voting scheme that is encoded in block headers. This approach offers a transparent and tamper-evident method for adding or removing signers [25]. Also, since only a limited, pre-approved set of authorities is allowed to create blocks, these nodes could theoretically collectively determine which transactions are included in the chain. This is because if the majority of them collude, they have the power to censor transactions (censoring signer) or reverse previous blocks (malicious signer) [25].

This review is not fully exhaustive. We primarily focus on PoW, PoS, PBFT and PoA because they are prominent in the literature that we reviewed. In addition to the mechanisms already discussed, other consensus protocols exist, including Delegated Proof-of-Stake, Proof-of-Capacity, Proof-of-Elapsed-Time, Proof-of-Burn, Proof-of-Weight, and BFT variants such as HotStuff. Each of these come with its own set of trade-offs, such as additional token economics, significant storage or computational needs, dependence on special hardware, or extensive inter-node messaging [13]. These should be considered for evaluation in future research.

**Findings and conclusions**

Of the mechanisms we investigated, Proof-of-Authority is the most appropriate choice for our system. This mechanism depends on a small group of pre-approved validators, which helps to decrease computational demands and energy consumption. Since we control the branches, this model fits our architecture. PoA reduces computational requirements and minimizes network traffic by eliminating the need for resource-intensive mining, which allows for real-time processing at the edge. The Ethereum Clique specification introduces an on-chain voting system for managing signers. This helps ensure a clear and tamper-evident method for adding or removing authorities. The PoA model is also energy efficient. However, it relies on the trust placed in the chosen authorities, which requires transparent processes for trust management. A problematic selection process or potential collusion among signers can compromise security and enable transaction censorship.

### 4.2.3 Theme 3: Privacy with multi-party computation

**Summary**

Multi-party computation, or MPC, allows multiple parties to collaboratively compute a shared function while keeping their individual inputs confidential. It can ensure confidentiality through the integration of secret sharing and also, in some implementations, mechanisms such as message authentication codes. Protocols, including the SPDZ family of protocols, incorporate an offline phase to transfer intensive computations away from the online process. Active security makes it possible to identify tampering, even in the presence of dishonest participants.

**Literature review**

MPC enables multiple parties, who may not trust one another, to work together in order to compute a function using their private inputs, all while keeping those inputs confidential from each other [7]. The guarantee of confidentiality, alongside the ability to analyze data collaboratively, makes MPC a promising option for secure and privacy-preserving data management [7, 40].

In scenarios that involve fault detection within sensitive IoT environments, where every device or branch contains sensitive information such as sensor readings or local fault scores, this data may need to be kept confidential at all times, even as the system calculates the outcome.

The concept and foundations of MPC has its origins in the foundational research conducted by Yao [41, 42], as well as the contributions from Goldreich, Micali, and Wigderson (GMW) [43, 44]. This established the groundwork for protocols designed for both two-party and multi-party scenarios, focusing on passive security measures against adversaries [43]. These early MPC protocols were able to provide security against these "semi-honest, passive" parties, however, real-world systems require protection against active attackers who may randomly deviate from the specified protocol [43, 44, 45].

Garbled-circuit methods, like Yao's protocol [41], enhance two-party computation by minimizing communication rounds. However, they require costly zero-knowledge proofs or cut-and-choose procedures for active security [20], which can make them less suitable for regular MPCs in environments with limited resources.

Making MPC more secure against malicious parties, while also speeding up runtime and providing useful tools that can be used in the real world may help MPC's utilization. One of the primary sets of protocols identified that assist in achieving these objectives is the SPDZ family of protocols [17, 18, 19]. For example, SPDZ protocols can include information-theoretic message authentication codes with each secret share. If any tampering occurs, it affects the MAC check and leads to the protocol's termination, thereby preserving correctness in spite of malicious actions [18, 19].

Beaver triples can help speed up MPC. These are generated in the offline phase and are simply hidden sets of supporting values that are created prior to the launch of the online MPC tasks, which lets participants bypass complex processes (as these are used as MPC operations are performed, as to avoid heavy work during these operations) when they need to process confidential inputs at a later stage [18, 20].

The MP-SPDZ [46] and Sequre [47] frameworks offer high-level compilers that convert annotated Python or domain-specific languages into optimized MPC protocols, which minimizes the cryptographic complexities for developers. This could speed up MPC development and integration, because it makes MPC more accessible. MP-SPDZ specifically allows users to set parameters such as the security guarantee and the algebraic domain (the ring size), which can be adjusted depending on various input-size needs. MP-SPDZ also can use an offline preprocessing phase to enhance overall efficiency [46].

Real-world implementations showcase the practicality of MPC. For example, MPC in healthcare allows for the collaborative analysis of patient data for research purposes, all while maintaining patient privacy [48]. For IoT specifically, Zhao et al. [7] performed an investigation into MPC in these environments, demonstrating that protocols that are well-designed can securely aggregate sensor readings across distributed nodes.

Interpreting the results from the study by Zheng et al. [4], the integration between MPC and blockchain can in theory further enhance the security, trustworthiness and reliability of data. In this system, blockchain's immutable, decentralized and verifiable ledger [4] could act as an enhancement to MPC,

which could help with securing and validating the results. This is because blockchain can store the output data (results) on-chain, providing an immutable audit trail for these results [4].

MPC presents promising solutions for IoT fault detection systems, like the one proposed in this thesis. However, there are several challenges that need to be addressed in order to reach its potential. To begin, many MPC protocols introduce noteworthy computational expenses on every participant involved [7]. Traditional online phases rely on high-precision modular field operations, for example, implemented via Montgomery arithmetic, to perform the necessary secret-shared additions and multiplications efficiently [18].

#### Findings and conclusions

Our analysis indicates that MPC maintains data privacy through the integration of secret sharing along with techniques such as MAC checks. Protocols like SPDZ incorporate information-theoretic MACs with each share and will stop functioning if any tampering is detected, guaranteeing correctness in the context of active attacks. Beaver triples that are created during an offline phase help to shift costly operations away from the online phase, which minimizes the runtime overhead for participants. High-level compilers such as the ones provided by MP-SPDZ and Sequre take annotated Python or domain-specific code and convert it into optimized MPC protocols, making it easier for developers to work with these technologies. Integrating blockchain with MPC enhances the security of computation outputs by recording them on an immutable ledger, which also offers an audit trail for the results.

### 4.2.4 Theme 4: Edge computing for real-time IoT fault detection

#### Summary

Edge computing helps with the processing and storage of data for devices placed on the network edge. This method reduces latency, saves bandwidth, and ensures that sensitive data remains on-site. Research indicates that it has the capability to gather data from numerous sensors in milliseconds and transfer demanding tasks to more robust edge servers. But, risks associated with physical access to edge devices require the implementation of end-to-end encryption and security measures.

#### Literature review

Edge computing is a distributed computing model that processes data closer to its origin (for example: in IoT settings this would be a device placed either physically on the same network, or close to the IoT source devices). This approach reduces the dependence on transmitting data to distant servers, which helps minimize delays and conserves bandwidth [10, 21]. This method moves cloud capabilities, such as computing and storage, to nodes at the network edge and is intended to meet requirements for low latency and high bandwidth, support security-sensitive applications, and offer more intelligent services in close proximity to data sources [21].

These features make edge computing an appealing choice for secure and fast real-time data processing [21], which can also apply in IoT systems where quick response times and data security can be important. The research highlights several advantages of using edge computing in this context.

One use of edge computing in this context is the study by Liu et al. [8], which integrate a secure MPC scheme for real-time data aggregation within a cloud-edge computing framework. This successfully moves most of the data processing responsibilities away from both the cloud and the limited industrial IoT (IIoT) devices. In this model, each edge server gathers raw data from its local group of sensor nodes, performs the aggregation and processing of the data, and afterwards sends only brief, region-level summaries to the central cloud [8]. This design reduces communication overhead by eliminating the need for numerous individual sensor transmissions to the cloud. It also decreases end-to-end latency, as edge servers are located closer to the devices and can process data in around 25 milliseconds for 100 nodes [8].

The study demonstrates that by maintaining raw measurements within the local network and sending only anonymized aggregates, the framework protects device privacy while avoiding the need for complicated encryption at the node level. In practical terms, when there are 20 to 100 IIoT nodes connected to each edge server, the system is able to perform real-time aggregation and computation tasks within a timeframe of single to double-digit milliseconds [8]. This performance likely satisfies the response-time requirements of industrial applications.

Edge computing also specifically enhances data privacy by allowing sensitive information to remain within localized systems. Processing data at edge devices reduces the risk of exposure that can happen during transmission to external servers [10, 21], which is particularly relevant for applications in sensitive domains like financial systems or critical infrastructure.

Edge computing can also lead to higher energy efficiency of the overall system. By transferring computational tasks to edge nodes, the energy consumption of individual devices can be reduced. Studies by Wadhwa et al. [9] and Nguyen et al. [29] emphasize the role of edge computing in achieving energy-efficient IoT operations.

Wadhwa et al. [9] suggest that the proof-of-work mining process of the blockchain could be offloaded to a single, high-capacity edge node, which would be selected based on its CPU, RAM, and bandwidth characteristics. Their design reduces energy consumption by approximately 21 percent and decreases memory requirements by 24 percent, all while ensuring that block-generation rates remain suitable for IoT networks [9].

Nguyen et al. [29] review, through studies in specific domains, how edge computing moves core IoT processing from the cloud to local nodes. The results from this change improve the system's responsiveness by reducing the time data takes to travel and decreasing the load on the network [29]. In the first study, in the healthcare sector, edge servers play an important role by preprocessing patient data directly on the network edge [29]. The study found this method could addresses real-time requirements while making sure that data remains on-site, and as a result safeguard privacy as well.

Another example appears in the study focussed on smart grids, where edge-hosted nodes placed close to generators and consumers speed up energy trading, enable quick demand response, maintain reliable audit logs, and reduce load on the network [29].

While edge computing offers many advantages, it also presents several challenges. To begin with, resource constraints may be a challenge in certain edge devices. Edge devices sometimes may not have enough computational power and storage capacity to manage complex processing tasks. To address this issue, Cao et al. [21] described edge architectures can carry out computational offloading. At first, there is a phase referred to as the offloading decision phase, which examines what tasks to offload, the extent of the offloading, and the appropriate location for this process. This evaluation takes into account various metrics, including energy consumption, latency, available bandwidth, and data size [21]. Afterwards, there is a phase dedicated to resource allocation, where tasks are assigned to one or more servers [21], taking into account the capacity of each server.

While edge computing decreases dependence on centralized systems and speeds up local processing, it can also expose certain security concerns. First, to begin with, having physical access to edge devices, which can be situated in remote or public locations, enables attackers to interfere with hardware components, such as by soldering or hardware attacks that exploit exposed ports (e.g. USB, PCIe), or to take advantage of side-channel attacks that leak data [49]. This can present a risk of data theft or potential device malfunction. Local cyber attacks can also be initiated, for example, attackers can also inject malicious firmware or carry out man-in-the-middle attacks on communication interfaces, enabling interception or modification of data [49].

To protect against these threats, it is important to encrypt data whether it is stored or being transmitted [50]. For example, cryptographic schemes at the edge help protect data, even in scenarios where an attacker might obtain physical access [50, 49].

**Findings and conclusions**

Our analysis indicates that edge computing is capable of meeting the real-time needs for IoT fault detection by handling data near its origin. According to the research by Liu et al. [8], edge servers were able to aggregate data from as many as 100 nodes in approximately 25 milliseconds. This performance meets the low-latency requirements typical of industrial applications. Maintaining raw measurements on-site also helps to protect data privacy and reduces communication overhead. Offloading strategies, as outlined by Cao et al. [21], enhance the functionality of limited edge devices. This process distributes tasks to servers with sufficient capacity. Research by Wadhwa et al. [9] also show benefits: they observed a 21% decrease in energy consumption and a 24% reduction in memory usage when proof-of-work mining transitioned to a single, high-capacity edge node. Similarly, Nguyen et al. [29] described similar improvements in efficiency and responsiveness within healthcare and smart-grid contexts.

However, having physical access to remote or public edge devices can lead to hardware tampering, firmware injection or side-channel attacks, as highlighted by Jin et al. [49]. Sheikh et al. [50] therefore suggest that to address these threats it is advisable to implement end-to-end encryption for both stored and transmitted data at the edge.

### 4.2.5 Theme 5: Security challenges in IoT

#### Summary

The distributed architecture of the IoT, along with its resource-limited devices and reliance on networks, can lead to vulnerabilities across the entire stack, making systems susceptible to attacks across multiple OSI layers[1]. The lack of uniform security protocols along with real-time threat detection can result in breaches within IoT networks.

#### Literature Review

The IoT ecosystem can be useful because of the range of applications it offers, but it also brings with it certain security challenges that need to be addressed. These challenges stem from its inherently distributed architecture, reliance on resource-constrained devices, and dependence on network communication, which can create vulnerabilities across the entire stack [31, 3]. These weaknesses make IoT systems susceptible to attacks, complicating efforts to ensure data privacy and secure communication, as seen in the study by Sasi et al. [52].

One of the concerns is the centralized architecture of traditional IoT systems, which introduces risks such as a single point of failure and reduced system reliability [3]. Security breaches in IoT devices can lead to a chain reaction throughout networks, which can further increase these risks [31].

In industrial networks specifically, the increasing integration of IoT sensors and controllers forms a significant expansion of the attack surface. By 2020, almost half of manufacturing companies had implemented industrial IoT, and the number of devices on factory floors is expected to more than double by 2025. Nevertheless, 82 percent of these companies have experienced a cyber breach related to IoT within the last two years [53]. This highlights the importance of implementing "security by design".

IoT devices often operate under significant resource constraints, with limited computational power and memory. These limitations make it difficult to implement strong security protocols, leaving devices vulnerable to attacks such as impersonation, eavesdropping, and denial of service [31, 52]. Hardware vulnerabilities can be another concern, especially given the lack of standardization in IoT device manufacturing [52].

In a context like banking, where unauthorized access can result in risks, encryption and real-time monitoring can be important for security [54]. Here, IoT endpoints such as RFID readers and biometric sensors are used. These devices can contain data that can be captured by IoT devices, which is useful for analysis [54], but these could also draw the attention of cybercriminals seeking to steal or modify this data.

In general, the risks extend across the IoT stack. In the OSI model, threats can span multiple layers. At the network layer (Layer 3), routing attacks and replay attacks can disrupt or eavesdrop on data flows [52]. At the application layer (Layer 7), attackers can intercept sensitive data or inject modified commands into IoT services [52].

Centralized IoT architectures can increase these risks by creating single points of failure. These vulnerabilities make centralized systems susceptible to data tampering and loss, which presents an important challenge in sensitive environments [31, 29]. Adding to these issues, existing IoT systems frequently lack real-time threat detection capabilities, delaying responses to attacks and increasing the potential damage [52].

[1]The OSI (Open Systems Interconnection) model is a conceptual framework that organizes network functions into seven distinct layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application. This structure intends to promote interoperability and help with the design of protocols[51].

**Findings and conclusions**

Our analysis indicates that IoT systems can have vulnerabilities across all layers of the OSI model. Devices with constraints often struggle to implement robust security protocols, making them vulnerable to threats such as impersonation, eavesdropping, and denial-of-service attacks. Centralized architectures can create single points of failure, which increases the risk of data tampering and loss. Examples from industrial deployments highlight the risks: A noteworthy 82% of manufacturers have reported experiencing an IoT-related breach within the past two years. Attacks can vary from routing and replay exploits to command injection and data interception. Without real-time threat detection, response times suffer and attackers have greater opportunities to inflict damage.

Therefore, implementing real-time monitoring is important for identifying anomalies early, preventing them from spreading further. For example, in banking, a faulty RFID reader could result in unauthorized access. Decentralized frameworks can eliminate single points of failure in systems that can detect these anomalies and enhance auditability. This can improve and maintain trust in IoT networks.

### 4.2.6 Theme 6: IoT in high-security environments

**Summary**

In the banking sector, IoT systems provide security to protect vaults. In the industrial domain, these systems can combine predictive maintenance with secure monitoring to prevent failures and unauthorized access. Some of the main threats to IoT and industrial networks are ransomware targeting critical systems, insider attacks, signal interception, and issues regarding privacy. Many organizations still do not implement complete network segmentation, which increases the probability of cross-network attacks.

**Literature Review**

This theme explores how IoT technologies can be integrated into environments that require increased security, such as in banking and critical industrial facilities. The aim is to identify the distinct challenges that exist within high-security environments. The literature reveals varied use cases for IoT in these settings, alongside challenges that must be addressed to ensure their effectiveness.

Ransomware has expanded its impact beyond only enterprise IT; it is currently an important threat to critical real-time systems. For example, in the healthcare sector, the number of monthly ransomware incidents more than doubled from 2021 to 2022, typically severely impacting both hospital IT and operational technology [53]. Attacks on utilities, like the 2021 Florida municipal utility Lake City hack [53], show that hacks can affect critical services.

In the financial sector, IoT systems fulfill an important part in preventing unauthorized access to critical areas, including vaults. These systems use real-time anomaly detection and alert mechanisms to handle breaches [54]. Multi-factor authentication, RFID, passwords, and facial recognition, makes sure that only authorized personnel can access sensitive areas. This process is facilitated by IoT, which, in this context, serves to improve both physical and digital security [55, 56]. The use of IoT-enabled monitoring increases these capabilities through the use of sensors within vaults to identify any tampering or physical breaches. For instance, devices that have vibration and motion detectors can constantly monitor the surroundings, sending alerts when there are indications of suspicious activity [56, 55].

It is important to take into account the challenges associated with implementing IoT in high-security environments. Complex threats, like insider attacks, where employees or contractors with access to secure systems may intentionally or accidentally compromise devices, and signal interception, which involves intercepting wireless signals to gain access to sensitive data or disrupt operations, highlight the challenges in safeguarding these environments [52], among other risks such as impersonation, denial of service and eavesdropping [31]. In addition, processing sensitive information, including personal biometric data and transaction data, can create privacy issues, mostly with regard to adherence to the strict data protection regulations such as GDPR [5].

An important case study to investigate in the context of IoT security within high-security environments is the Stuxnet cyberattack. This incident illustrates the possible impacts of IoT vulnerabilities in settings that require high security. The now well-known worm was aimed specifically at programmable logic controllers (PLC's) within Iran's Natanz nuclear facility, and managed to get past air-gapped networks through the use of USB drives. Stuxnet altered the speed of centrifuges involved in uranium enrichment,

resulting in physical damage and disruption of nuclear enrichment operations, which was the intended consequence of the worm [57]. This attack demonstrated how cyber tools can affect critical infrastructure, even when systems are physically isolated. It also illustrates the challenges involved in securing industrial IoT systems and the lengths to which attackers may go to accomplish their goals.

Adding to this problem is the fact that just 24 percent of manufacturers have achieved full network segmentation between IT and the operational technology systems [53]. The lack of separation enables attackers to transition from enterprise systems onto operational networks, which could lead to disruptions.

#### Findings and conclusions

Our analysis indicates that IoT systems used in high-security environments enhance both physical and digital access control. This is achieved through real-time anomaly detection, multifactor authentication methods, RFID, passwords, and facial recognition, as well as sensors that monitor for vibrations and motion to alert on any tampering attempts. These measures are designed to protect vaults and essential infrastructure; however, they can experience complex threats like insider attacks and the interception of wireless signals. Ransomware has become more focused on operational technology, evident from the rise in incidents within the healthcare sector and the 2021 Lake City utility hack. Also, the Stuxnet worm illustrates how attackers might even be able to enter air-gapped networks, leading to potential physical damage. Only 24 percent of manufacturers have achieved complete IT-OT network segmentation. This lack of separation leads to lateral attack paths that are vulnerable.

## 4.3 Discussion

The thematic review researched three technologies, blockchain, MPC, and edge computing, that as a whole seem to be able to theoretically address the need for real-time fault detection and data privacy in high-security IoT environments. This section explores the findings and discusses them in relation to the design goals stated in this thesis.

To begin with, the immutable ledger of blockchain offers an effective guarantee of log integrity, which is in line with the need for tamper-evident audit trails. However, the existing literature clearly indicates that traditional proof-of-work chains experience significant latency and energy expenses, particularly in environments with limited resources.

Lightweight consensus protocols like proof-of-authority offer a practical solution that preserves immutability while minimizing the computational requirements associated with proof-of-work. This indicates that a private chain based on a lightweight consensus mechanism offers the most practical blockchain-based solution for secure logging in an IoT edge deployment.

Our analysis further indicates that implementing a hybrid on-chain / off-chain storage model can be important for balancing the immutability of blockchain technology with the data-erasure requirements by regulations such as the GDPR. This would allow for both auditability and legal compliance.

We also found that MPC improves privacy by allowing multiple parties to collaboratively compute a function using their private inputs while keeping those inputs confidential. Our review shows that protocols providing active security, such as SPDZ, come with overhead in terms of both computation and communication, and therefore may necessitate the use of more powerful systems.

Finally, edge computing places processing near the data sources, which helps to lower both latency and bandwidth usage. Research indicates that edge servers are capable of aggregating and preprocessing sensor data in just a few milliseconds in some cases, all while maintaining privacy by keeping the raw measurements stored locally.

In our architecture, this could mean that each branch's edge node should also serve as the blockchain validator, and a MPC party. This setup could help with localizing all sensitive operations and reducing the risk of data exposure across wide-area links, while making such a system possible. This design will be researched in this thesis.

The research we reviewed indicates that the proposed framework is feasible. When combined, these three technologies have the potential to provide a complementing system: edge computing provides low-latency processing, MPC safeguards inputs during score aggregation, and blockchain stores the hashes of the data.

The main limitations are related to resource overhead (blockchain nodes require storage and validation cycles, whereas MPC uses computational resources and network bandwidth). The PoA model relies on trusted validators, which introduces a trust assumption and risk of collusion. The edge computing layer requires physical security measures to prevent tampering and ensure device integrity. These findings point to the need for a threat model that covers validator trust and edge node compromise.

# Chapter 5

# System design

## 5.1 Introduction

This chapter provides a detailed overview of the architecture for the fault-detection system, starting with the main components and how they interact in the process of detecting, correlating, and documenting faults in IoT devices. The major design choices and the reasoning that supports them is described in section 5.3.

## 5.2 High level system architecture

The main components of the system and their interactions are illustrated in Figure 5.1, along with the data flows between the components. The system operates through a collection of Docker containers, with each one executing a specific function in the fault-detection process:

1. Each branch runs an edge service designed to handle faults. In this proof-of-concept, we use a tool called the IoT simulator, which selects this severity score. This tool allows users to create and test virtual IoT devices in a controlled environment, which aims to replicate the functionality of sensor devices. The process starts with picking a branch and sublocation, choosing a fault type and assigning a severity score. Each event is sent as a JSON payload to the Coordinator API. Each event is also timestamped when it is received by the service.

2. When the IoT simulator service sends a fault, it sends this to the coordinator, a service that is present at every branch. The Coordinator obtains unprocessed fault events. It marks the time of reception, and logs that score. This data is encrypted using AES-CBC, then written to a local off-chain log, and a SHA-256 hash of the fault data is computed. The coordinator service transmits that hash to the blockchain API, where it is stored as an immutable transaction. The MPC node at that same branch also has access to a copy of this fault data, which is used during the MPC process.

3. When a fault is detected, or when specifically requested, we can initiate a threat update across multiple branches. When this is requested, the coordinator service reaches out to the local MPC node, which holds the severity sum and event count for its branch. All MPC nodes run MP-SPDZ simultaneously. They calculate an overall average severity without revealing the inputs from the branches.

4. After the completion of the MPC-process, all nodes reveal a threat percentage (a weighted average between the branches) to each branch operator, indicating the level of global threat. One node gathers the results, hashes them, and then submits the score to the blockchain service. This transaction stores the network threat score on the blockchain, in a temper-evident way. Every branch has the ability to query the chain or retrieve their own local MPC result in order to get the most recent score and modify its local security policies as needed.

5. The MPC protocol does not reveal raw sensor readings or branch scores. Rather, each branch divides its private input into random secret shares and only exchanges those shares. Individual

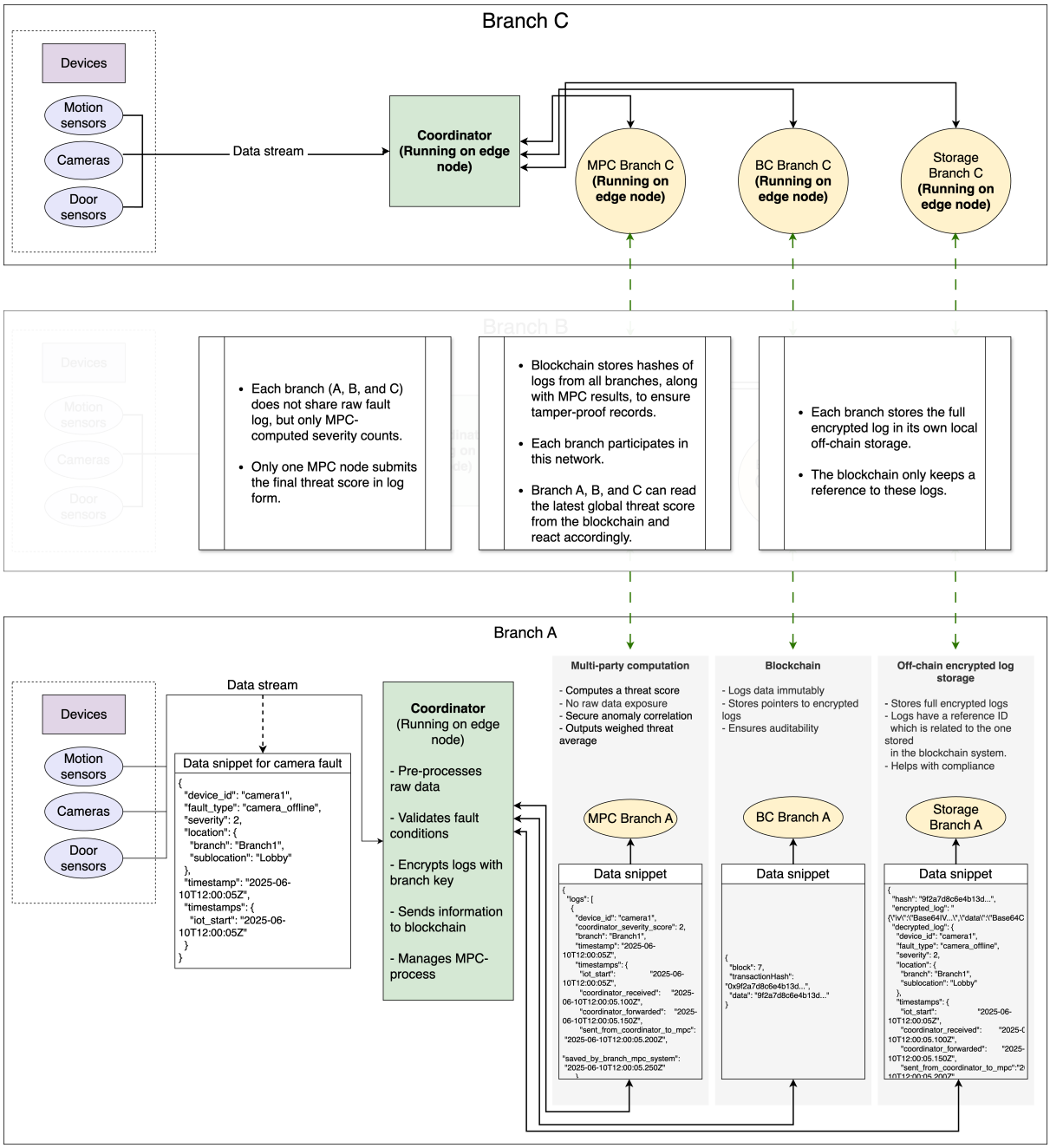



Figure 5.1: High-level system architecture of the proof-of-concept prototype for fault detection. In this model, branch A is the branch submitting the mpc calculated outcome to the blockchain. The MPC, Blockchain (BC), and Storage services run as services on an edge system running at each branch.

shares do not disclose any information on their own; it is only when all shares are brought together that the value becomes visible.

6. Every branch operates its own blockchain node (the blockchain service) alongside the other services. The node relays transactions to other validators, which assist in the maintenance of consensus. Local off-chain storage retains the complete encrypted logs and help with compliance, for example, when needing to remove data in response to deletion requests, such as those under GDPR, like a "right to erasure" request. This approach also helps maintain a compact on-chain storage size, since less data is stored on-chain.

7. The on-chain ledger only contains SHA-256 log hashes as well as threat-score details. The complete, full, logs remain encrypted off-chain which allows branches to delete or archive detailed data (according to a set schedule or manually). This separation allows the system to comply with policies while maintaining a verifiable audit trail.

## 5.3 Design choices and rationale

In this chapter, we discuss the decisions for the design of the proof-of-concept system. Every section provides a rationale for one significant decision, utilizing insights from both academic literature and practical factors. Whenever we can, we ground the decisions in previous research, while primarily concentrating on specific engineering requirements.

### 5.3.1 Choice of number of branches

In our prototype, we restrict the branch setup to three branches, which consequently limits the MPC and blockchain parties to three as well. This decision allows us to perform the implementation and experiments within the limits of our test hardware resources, while still engaging in a multi-party computation and blockchain protocol. Three participants represent a manageable and practical deployment, like a head office working alongside two branches, while avoiding the added coordination challenges that larger groups would entail. We choose this minimal configuration to showcase the end-to-end functionality and performance in our environment.

### 5.3.2 Choice of programming language and key libraries

We choose Python because it offers quick development, an easy-to-understand syntax (Python adheres to the "simple is better than complex" core philosophy), and a well-established ecosystem [58]. The dynamic typing and concise nature of Python assist in reducing redundant code, allowing for rapid prototyping of complicated functions [59]. The widespread use of this technology in both research and the industry guarantees that it will remain sustainable in the future [60].

We choose to incorporate the following libraries into the codebase:

- Flask is used for building HTTP APIs and lightweight dashboards, making it suitable for demonstration purposes. The minimal core and the extension system enable the creation of microservices. This architecture separates functionality, such as API endpoints, dashboards, and MPC orchestration, into distinct services that can be developed, deployed, and scaled independently.
- We use the requests package to call other services via REST because its API is straightforward and synchronous, which helps in reliably coordinating steps across services.
- Pycryptodome is used for encryption, and hashlib for SHA-256 hashing. These ensure the security of off-chain logs and produce fixed-length digests for storing on-chain.
- Web3.py[1] is used for interacting with Ethereum JSON RPC[2], which is a way for developers and users to communicate with an Ethereum node, making it easier to sign and submit transactions to Besu[3].

[1] Web3.py is a library in Python that allows users to interact with Ethereum nodes through JSON-RPC.
[2] JSON-RPC is a protocol for remote procedure calls that operates in a stateless manner, utilizing JSON over HTTP.
[3] Hyperledger Besu is an open-source Ethereum client that supports permissioned proof-of-authority networks.

- python-decouple is used for managing configuration settings. It allows us to import configuration variables from the docker-compose files, making changing these easier instead of having to hard-code these within the scripts.
- The schedule library (in the IoT simulator) is used for periodic fault injection.
- The eth-account library is used for managing Ethereum accounts and signing transactions. It plays a role in both the blockchain setup and the web interface, facilitating the creation and signing of transactions.
- The eth-keys library is used via eth-account for low-level key management and for converting between raw private and public key formats.
- Psutil is used to obtain host system information. It is used in the data collection script to collect info about the CPU, memory, and other host features.
- The Docker package[4] is used in the data collection script to gather active containers, gather size and performance metrics.
- The pytz package is used for timezone handling.
- Supporting packages include standard-library modules such as json, os, time, threading, logging, csv, argparse, and datetime, which are used for scripting and/or input/output operations.

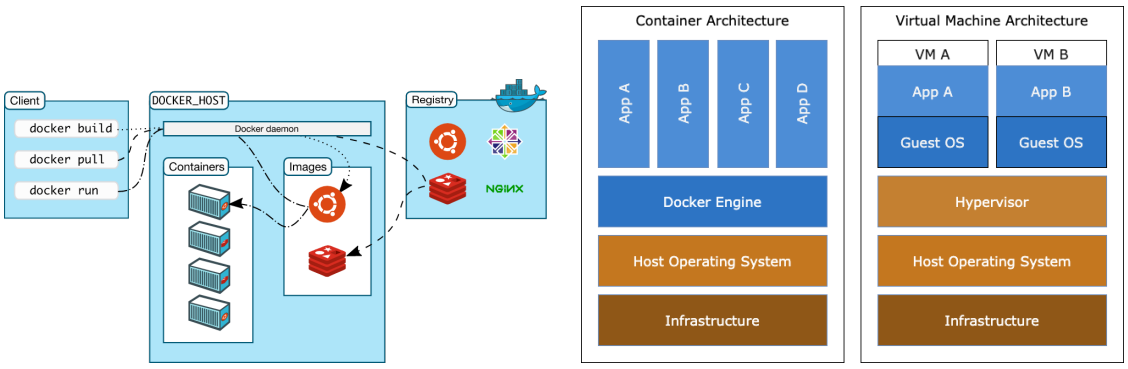
### 5.3.3 Microservices architecture

This design isolates failures, ensuring that, for example, a bug in the simulator does not lead to a crash of the MPC service. It also allows for the ability to develop and test services simultaneously. Deployment and upgrades are easier because we can introduce a new version of a single service without the need to rebuild the entire system. And finally, this design also accommodates future modifications within the system, such as testing changes proposed in the future work section.

### 5.3.4 Containerization with Docker

Each service is packaged in a Docker container, which combines its application code, runtime, libraries, and configuration into one single image [61]. This guarantees that the service operates in the same way, regardless of whether it is on a developer's laptop, a cloud server, or an edge device.

Docker containers make use of the host operating system's kernel, which allows them to avoid the complete virtualization stack typically associated with virtual machines. This lightweight approach provides isolation between containers and the host processes, resulting in CPU and memory performance close to native execution [61]. This difference is illustrated in Figure 5.2, where Figure 5.2b shows the main architectural differences between the Docker architecture and a virtual machine architecture.




(a) Docker architecture overview [62]. (b) Containers vs. virtual machines [63].

Figure 5.2: (a) The Docker workflow: client commands, daemon, images, containers, and registry. (b) Comparison of container and VM architectures.

[4]Docker is the official Python SDK for Docker, enabling programmatic control of containers and networks.

- A Docker container is created using the `docker build` command. The client sends a Dockerfile to the Docker daemon, which then processes each instruction to build a series of immutable filesystem layers. The layers that make up a Docker image can be found in the "Images" panel shown in Figure 5.2a. When you execute `docker run`, the daemon takes those read-only layers and adds a thin writable layer on top, creating the running container.
- Every image consists of an operating system, a language runtime (specifically Python), along with particular versions of libraries. When we specify exact package versions in `requirements.txt` and use Docker image digests[5], we ensure that each time a container is launched, it operates with the same binaries and dependencies.
- Containers operate within distinct environments. For example, it is possible for one service to use Python 3.11 while another operates on Python 3.9, and they can do so without any conflict. When you upgrade or add a library in one container, it does not impact the other containers.
- We can create images for various CPU architectures, such as x86_64 and ARM. A single container image can operate on various hardware platforms without requiring many, or even any, modifications to the service code.
- Docker allows us to "lock in" the dependencies, which allows possible future research to reuse the exact environment we delivered.
- Docker provides metrics such as CPU usage, memory consumption, block I/O, and network I/O through the `docker stats` API. These metrics focus solely on the activities within the container, eliminating any interference from other host processes or the overhead associated with virtual machines. We use this API to gather container metrics[6].

### 5.3.5 MPC framework

We aim to use a framework that integrates active security measures and offline preprocessing, while ensuring that the codebase remains maintainable. We use MP-SPDZ by Keller [46], which is a C++-based framework that offers support for various protocols, including SPDZ$^{2k}$. It comes with detailed documentation, example programs, and an active community on GitHub, with over 1100 stars, 57 contributors and 324 forks [64]. The software is actively maintained, which adds to our choice. MP-SPDZ features a configuration-driven approach that allows for the adjustment of parameters such as the ring size, security level and the number of parties, while the performance figures for comparable protocols published in the paper by Keller [46] surpass those of pure-Python frameworks, such as MPyC.

Within MP-SPDZ, we implemented the SPDZ$^{2k}$ protocol, which merges additive secret sharing over $\mathbb{Z}/2^k\mathbb{Z}$ with information-theoretic MACs to identify tampering, even when faced with a dishonest majority. We seek this behavior because, in high-security environments, we assume it is most important to maintain correctness even when more than half of the parties may collude[7].

In addition, MP-SPDZ offers the ability to switch between different protocols without the need to rewrite the application logic much. The numerous examples and comprehensive API reference make it easy to integrate with the containerized services.

While we implement SPDZ$^{2k}$ for now, MP-SPDZ allows for the transition to other protocols in the future, such as classic SPDZ over a prime field or semi-honest GMW. If a different trust assumption or field is more appropriate for a specific use case, the necessary infrastructure is already established.

### 5.3.6 Web framework

We use Flask to expose REST endpoints and serve simple UIs. Its advantages are as following:

- The clear routing and request handling in Flask allow us to measure timing accurately for performance metrics

[5]The digest refers to the SHA-256 hash derived from the image manifest JSON. This method provides a unique identification of the specific contents of an image, including its layers, metadata, and configuration. As a result, retrieving an image by its digest (for instance, `repo/mpc-mpc_node_0@sha256:abcd1234...`) ensures that you always get the exact same image, regardless of any changes to its tags.

[6]Please see appendix B.4 for the implementation.

[7]If we were to assume we have an environment where most parties are honest, we could opt for a protocol that relies on this honest majority, which can decrease communication costs.

- Both the core API and the Jinja2 template engine (loading the HTML files) are lightweight.
- Plugins are available for logging and serving static files.

### 5.3.7 Blockchain engine

We select Hyperledger Besu as the blockchain engine due to its native support for a permissioned Ethereum network operating in proof-of-authority mode, specifically Clique. This allows us to operate a private chain in which only pre-approved nodes have the ability to seal blocks.

Besu offers a comprehensive JSON-RPC API, including web3, which we use via Web3.py in the Flask service. The standard interface allows the coordinator and MPC components to submit transactions, query blocks, and monitor events without the need for custom protocol code. In the setup scripts, we establish the network's genesis parameters and the configuration for static nodes. This ensures that each branch starts with the same chain rules and validator keys.

There is an active community surrounding Besu, along with well-organized documentation, which are also important aspects to consider during development.

### 5.3.8 Hybrid on-chain / off-chain storage

In the design, we avoid placing the actual log data on the blockchain; instead, we use a fixed-length "fingerprint" (a SHA-256 hash) for each entry. The complete logs, which are encrypted using AES-CBC, remain off-chain and are stored locally. When a request is made to erase data under GDPR, this system can delete the necessary off-chain data. Because the on-chain data only contains the hash, the contents of the data can no longer be recovered. Auditors can still confirm that a record once existed and can verify it later if it is restored, but no private information is left on the chain itself.

We make use of SHA-256 since it generates a consistent 256-bit output, which acts as a one-way fingerprint for the data. It is practically impossible to revert the hash back to its original plaintext form. SHA-256 is efficient and gets broad support from standard libraries, which helps ensure that auditors can verify event integrity with both compatibility and long-term stability in mind.

AES-CBC is what we choose for the encryption security because AES is a cipher approved by NIST, and has extensive support for hardware acceleration, making it fast. Using CBC mode with a new random IV for each message guarantees that even identical plaintext blocks will result in different ciphertexts. And, AES-CBC is widely supported in standard libraries, which makes it easy to implement and ensures strong confidentiality protections.

### 5.3.9 MPC parameters

We set up SPDZ$^{2k}$ within the ring $\mathbb{Z}/2^{128}\mathbb{Z}$, which is a 128-bit ring. The largest non-negative value before wrap-around is $2^{128}-1 \approx 3.40\times10^{38}$. We use non-negative integers only. Each party supplies two integers: (1) a scaled severity sum and (2) an event count. For each event, the severities range from 0 to 5. Before input, we scale the local sum by $10^6$ to get six decimal places. Therefore, across all parties,

$$\mathsf{total\_sev} \le 5 \cdot \mathsf{total\_events} \cdot 10^6.$$

We securely incorporate the contributions from all parties to calculate $\mathsf{total\_sev}$ and $\mathsf{total\_cnt}$. We then calculate the average through a single secure division using 24-bit precision (`int_div` with parameter 24) and reveal only the final result. Outside of MPC, we divide by $10^6$ to reverse the scaling and then normalize by 5 in order to express it as a percentage.

Even with a conservative estimate suggesting that the division routine might temporarily scale by $2^{24}$, we still stay within the 128-bit wrap-around limit provided

$$5 \cdot \mathsf{total\_events} \cdot 10^6 \cdot 2^{24} < 2^{128},$$

which is valid for any practical workload.

In our preprocessing, we utilize 64-bit statistical security, denoted as $-S$ 64. For this to work, $k$ must exceed the security parameter, which means that a 128-bit ring is the smallest standard option that accommodates this configuration along with our fixed-point headroom.

### 5.3.10 Simulated IoT devices

We create simulations of IoT devices to enable precise control over each test and ensure exact reproducibility, while also not needing actual hardware. The simulator is capable of generating faults at adjustable intervals with full control over the fault details, which allows us to evaluate the system under a variety of loads. Identifying real sensor failures falls outside the scope of this thesis, so for instance, a security camera that has analytics integrated in can identify a blurry image as a problem and report it to the system.

## 5.4 Low-level design

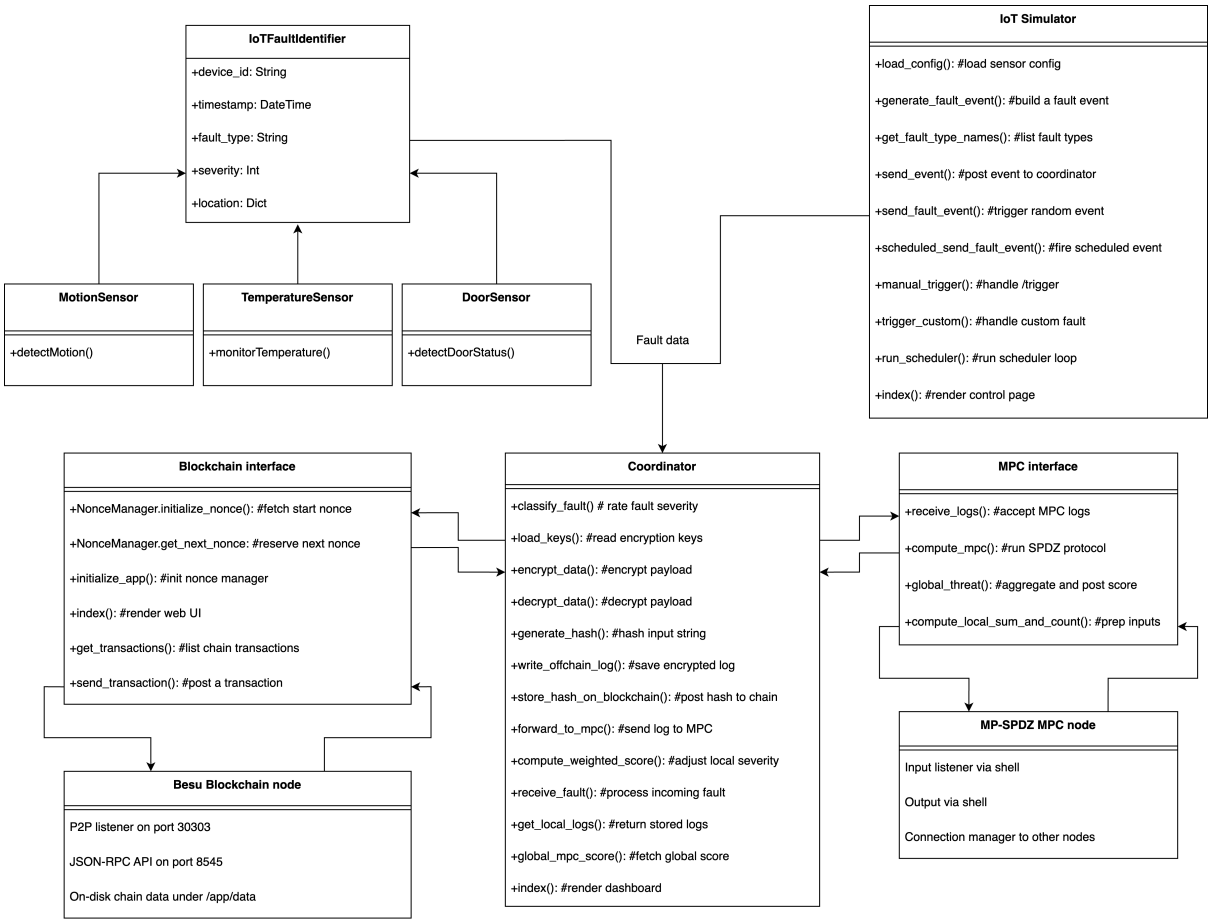



Figure 5.3: Low-level system architecture of the proof-of-concept prototype. Some helper functions are omitted. The IoT fault identifier (not implemented in the PoC) would be the module classifying and processing the faults from the IoT devices. This is implemented using the IoT simulator for testing purposes.

The prototype shown in the low-level design (Figure 5.3) divides functionality into containerized services that interact through a single Docker network (`iot_network`). Every service uses a REST API which send/receive payloads. Each service shown in the design has the following functions:

- **The coordinator service** operates on port 4000. The system uses AES-CBC for encrypting logs. The coordinator also calculates SHA-256 strings for each complete log and then submits these to the blockchain using Web3.py.
- **A Blockchain web interface** operates on port 3055. A Flask wrapper that interfaces with Hyperledger Besu is responsible for managing transactions. The graphical user interface is illustrated in Figure 5.4 and shows the transactions placed by the system, accessible through this service.
- **MPC nodes** operate on port 5000. Every container includes precompiled MP-SPDZ binaries and shared libraries located in the `/mp-spdz` directory, which are compiled before "assembling"

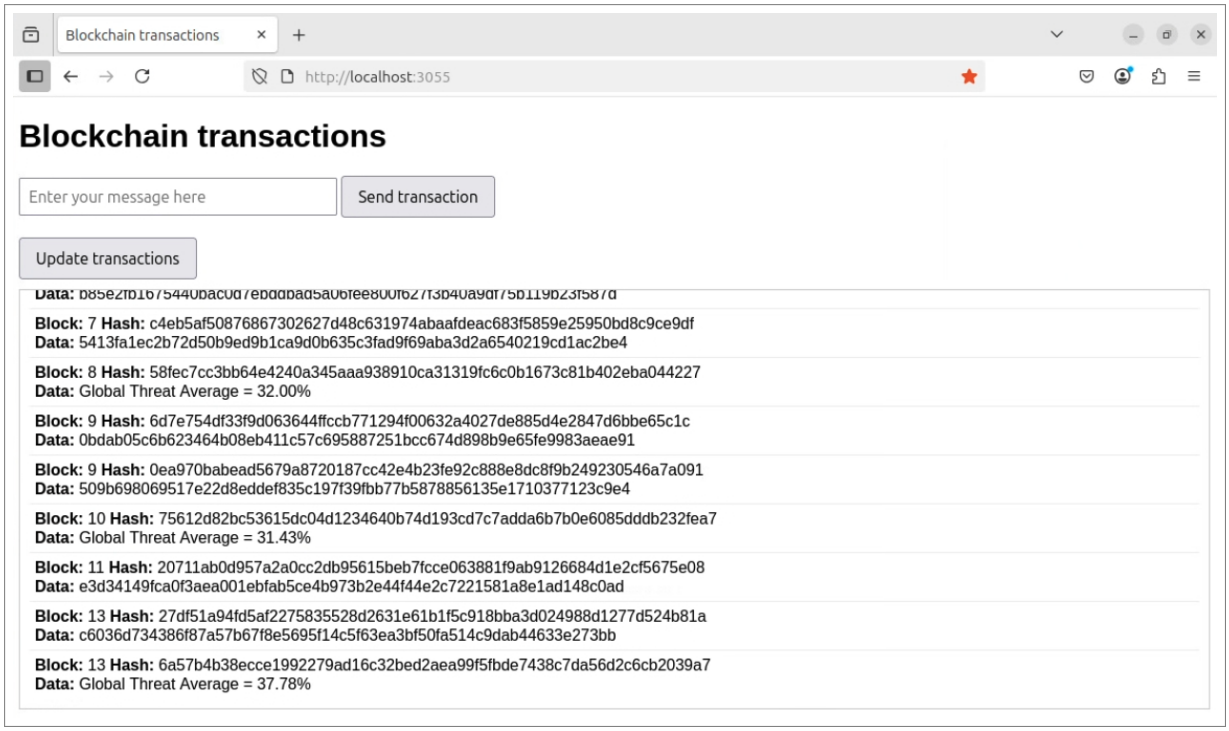



Figure 5.4: The blockchain user interface

the containers [8]. When we first run `Fake-Offline.x` [9], it creates Beaver triples in a versioned `Player-Data` folder. When running the containers (online phase), the `spdz2k-party.x` operates in interactive mode with a 128-bit ring and 64-bit security. The Python wrapper processes local logs from a JSON file, adjusts inputs for fixed-point arithmetic, and interprets the numeric output to calculate a percentage (the threat score).

The `/global_threat` endpoint in the coordinator container invokes `/compute_mpc` (the call used to start the calculations in the network) across all parties simultaneously. These nodes connect with each other, calculate the MPC score[10] and then check for consistent results. The results are sent to the blockchain.

- **The IoT simulator** runs on port 4100. The system simulates sensors ( IoT devices). The graphical user interface is illustrated in Figure 5.5 and shows the controls that can be used to send (custom) fault events. The service also allows for automated, preconfigured inputs though an API.

Every service uses host volumes to ensure that data and configuration are persistent. The dockerfiles are designed to install only the necessary OS packages and Python dependencies, which helps to keep the image size to a minimum. Services use Docker's integrated DNS, allowing them to communicate with each other using container names. The layout makes it easier to scale or replace individual modules and it also isolates these into distinct, testable components.

## 5.5 Data and evaluation plan

We assess the prototype based on key elements, which are performance overhead and scalability. We analyze four configurations, baseline (no BC, no MPC), blockchain only (BC on), MPC only, and the full system (BC + MPC), to isolate the impact of each component. We measure the power consumption in watts to evaluate energy efficiency.

[8] See Appendix C.2 that details this process.

[9] This MPC setup, which utilizes `Fake-Offline.x`, is intended only for demonstration purposes. The Beaver triples that are produced are generated offline and lack cryptographic security guarantees, so they should not be used in a production environment. For a successful deployment, it is important to have a secure and randomized offline preprocessing phase.

[10] See Appendix C.4 for the program that the MPC nodes run.

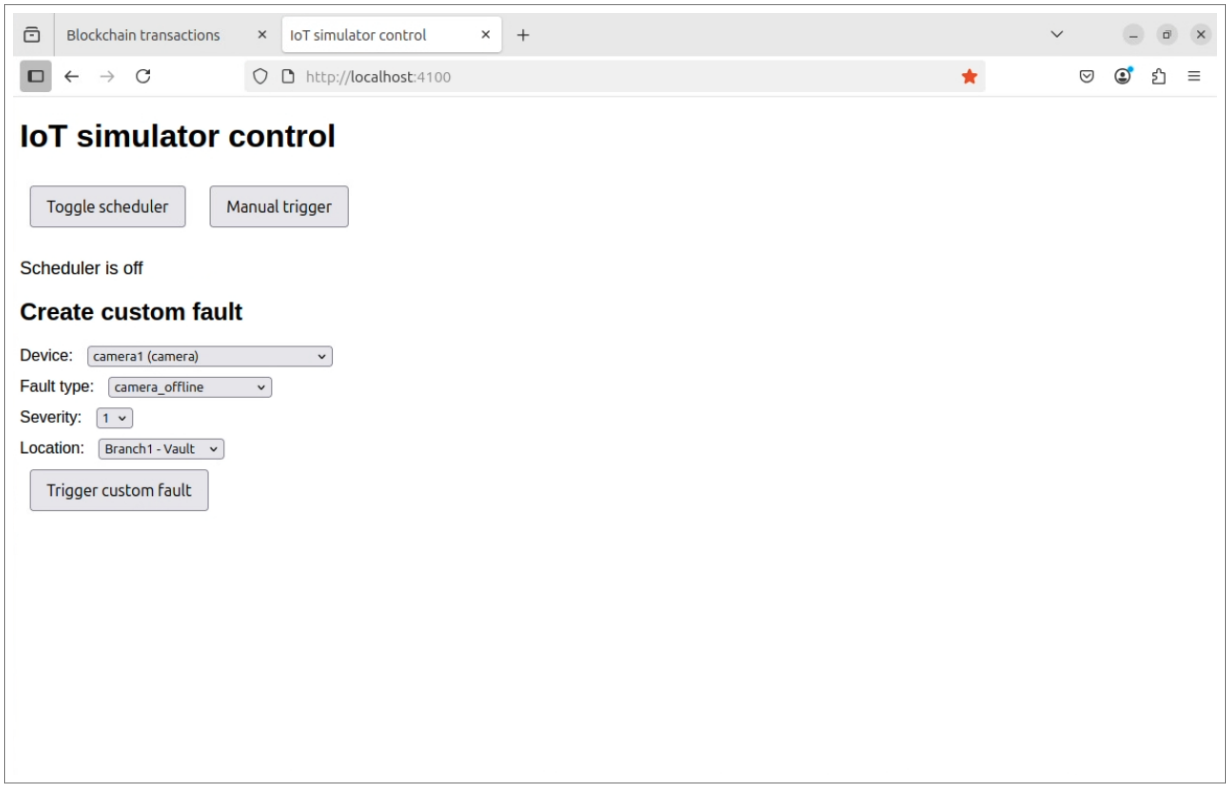



Figure 5.5: The IoT simulator web interface

### Experimental setup

All services are deployed on a single virtual Ubuntu-based Linux server. For details regarding the VM and host, see Appendix B.1. We carry out an experiment on fault sequences, varying the number of events. This allows us to observe the scaling of the total execution time (latency between components), throughput, and resource usage. Between runs, we clear all logs and delete the existing containers to start from a clean state. We also delete the volume mounts from the host system[11].

### Power measurement

We connect the server's power supply to a smart power meter[12]. We sample the average power draw during each experiment. We report both the absolute draw (in watts) and the delta relative to the baseline configuration.

### Metrics

During each experimental run, we gather the following data:

- **Network traffic (in MB)**. This refers to the total amount of data sent and received (also called the communication cost) by each MPC node.
- **Total execution time (latency between components in ms)**, the duration from the moment a sensor fault is injected until it receives confirmation from the blockchain, categorized by component (coordinator, MPC, blockchain).
- **CPU and memory usage**, the CPU seconds and memory (in MB) at the container level.
- **Storage used**, the storage used on the disk (in MB) at the host level, per container.

[11] We systematically stress-test the prototype by utilizing the `generate_sequence.py` tool (please see to Appendix B.3) to create fault-event CSVs across different load conditions.

[12] We use the INSPELNING power meter from IKEA. The data is collected and stored by Home Assistant, a service which can collect and store this data, which is running on a different server (not connected to the same power meter, as to not alter the results).

- **Power draw (Watts)**. Power draw is measured at the wall socket using a smart power meter connected to the server, and sampled at a specified interval by a server. Before running the containers, a baseline is recorded, and the power draw is determined by calculating the delta between these values to isolate the wattage of the container.

### Analysis

In line with our data collection plan, we collect metrics including traffic, latency, CPU usage, memory usage, storage, and power draw across three runs for both the baseline and full-system tests at each scale. From these, we can conclude:

- Network traffic in MB sent between MPC nodes.
- The latency difference between configurations.
- The extra amount of disk space used.
- The extra CPU-seconds used.
- The extra memory used.
- The additional watts used on average, compared to baseline.

Next, we will visualize and engage in a discussion about the findings. All figures and insights can be found in Chapter 7, along with direct connections to the research questions.

# Chapter 6

# Implementation

This chapter outlines the specific steps we took to develop the system architecture into a functioning prototype[1]. The focus is on understanding the mechanisms behind service integration, container orchestration, configuration management, and the workflows that are responsible for assembling, launching, and testing the system. When applicable, we refer to the relevant scripts and configuration files in the appendices for comprehensive details.

### 6.0.1 Service integration and container setup

Every component is contained within its own Docker container[2], all of which share a common virtual network called `iot_network`.

Within the coordinator container, which should be present at all branches in a fully running system, the entrypoint script looks for existing encryption keys in `/app/data/config`. These keys are used to encrypt the off-chain logs. The Flask application listens on port 4000 and sets up a host directory to store both encrypted logs and decrypted demo logs for persistence. Environment variables provide the blockchain RPC endpoint along with the three branch-to-MPC URLs.

The blockchain setup includes four containers: three nodes dedicated to Besu (the blockchain itself) and one node for the web interface. This web container, like the coordinator container, should be present at all branches. Before starting the system, a Python script managed by the Docker host creates the genesis configuration, the static-nodes list, and the validator keys, which are then stored in the `blockchain/data` directory. When a Besu container starts up, it copies the necessary files into its data directory, uses an initialization script to resolve peer hostnames to IP addresses, and then proceeds to launch Besu with the specified Clique parameters. The web interface container uses credential files to communicate with Besu, sets up its nonce manager, and provides REST endpoints on port 3055 for `/send_transaction` and `/get_transactions`, used for interfacing with the blockchain.

To set up the MPC service, we begin by compiling the MP-SPDZ artifacts within a specialized "builder" container. The builder begins by cloning the MP-SPDZ repository. Next, the `spdz2k-party.x` and `Fake-Offline.x` binaries are compiled for use in the container. This process also performs the preprocessing phase to generate Beaver triples. The outputs are exported to `mpc/compiled_mpc_bins`. The runtime containers subsequently mount these bins and libraries into `/mp-spdz`, along with the `Player-Data` folder for each node. Each container's Flask wrapper manages the calls to `/logs`, `/compute_mpc`, and `/global_threat`, using `subprocess` to invoke the SPDZ binaries and then parsing the output into JSON format.

The IoT simulator container is designed to load a JSON file that includes information on sensor types, fault definitions, and branch sublocations. A background thread, utilizing the `schedule` library, submits randomized events at a set interval. At the same time, the Flask application running on port 4100 allows for manual triggers, API control and features a simple web user interface.

[1] For an overview of the prototype, please see Appendix A.2.
[2] See Appendix A.1 for Dockerfiles and build scripts.

### 6.0.2 Orchestration scripts

We simplified the process of environment provisioning by using two shell scripts. The script `run_all.sh` is responsible for detecting (or when not present, creating) the Docker network. It triggers the MPC build only when necessary, generates the configurations for both the blockchain and the coordinator, and subsequently launches each service in detached mode within its respective subdirectory using the command `docker compose up -d -build`.

The script `run_specific.sh` offers a numbered menu that allows users to select specific services. It then goes through the same configuration steps solely for the selected options before launching them. This can be used when changing the files in or rebooting the containers. Both scripts use `set -e`, which ensures that any failure will cause the process to abort early, which avoids partial launches.

### 6.0.3 Continuous integration and repeatable builds

Only pinned (tested) versions of system packages and Python modules specified in a `requirements.txt` are installed by each Dockerfile to guarantee repeatability. To avoid downloading build tools into the final runtime images, we employ multi-stage builds for the MPC builder. The generate_configs scripts replace any outdated data before to each run, and all created artifacts (the aforementioned genesis information, node keys, and encryption keys) are created.

### 6.0.4 Logging, monitoring, and health checks

Every Flask service implements a basic health endpoint (`/health`) that, when alive, returns a status of 200. Docker's logging driver captures logs published to STDOUT, allowing for real-time analysis using `docker logs`. We capture both web3 and raw RPC faults for the blockchain web interface. And, to help diagnose shared-library and file-path issues when testing new binaries or versions of MP-SPDZ, we produce detailed stdout and stderr from the SPDZ binaries in the MPC wrapper.

### 6.0.5 Data-collection process

`collect_data.py` coordinates the controlled experiments on the host computer (the Docker host)[3]. The script queries the coordinator, blockchain interface, and Docker daemon for logs, transactions, and container statistics. It also schedules fault triggers and calls `/global_mpc_score`, calculating the MPC score. Docker statistics are sampled at a set interval, which includes memory consumption and CPU seconds. The script generates a host information JSON file and four time-aligned CSV files (coordinator report, blockchain report, Docker metrics, and MPC report) following a configurable cooldown period. Repeatable, time-stamped data for the overhead and performance analysis in Chapter 7 are provided by this process.

[3]Detailed data collection methodology and script extracts are in Appendix B.

# Chapter 7

# Results

This chapter discusses the findings from the controlled experiments. We created three fault-event sequences and tested each sequence across four different configurations: the baseline (which includes neither blockchain or MPC), blockchain only, MPC only, and the complete system that incorporates both blockchain and MPC. We executed each run using the `automate_collect_data.sh` script. This script initiates the services, introduces faults from the CSV files, and calls `collect_data.py` to collect coordinator logs, blockchain transactions, Docker statistics, and MPC measurements. Next, we will examine the four modes by looking at various factors such as latency, throughput, resource usage, network traffic, storage, and power consumption.

## 7.1 Sequence generation test setup

We systematically stress-test the prototype by ingesting fault-event CSVs under different load conditions, utilizing the `generate_sequence.py` tool, which generates these fault-event CSVs. We examine both the number of events and the overall time window, so we can analyze how latency, throughput, and resource usage change in relation to scaling. In Appendix B.3, a clear explanation is given on how `generate_sequence.py` processes these inputs and produces a sorted CSV of time-offset events, which are used to test the system. As shown in Table 7.1, the sequence file provides the exact inputs used in the code (time offsets, locations, severities, and MPC trigger flags), making later latency and throughput measurements reproducible.

| time_offset | location | device_id | fault_type | severity | sublocation | trigger_mpc_calculation |
|---|---|---|---|---|---|---|
| 799.000 | Branch1 | temperature2 | temperature_spike | 2 | Parking-lot | false |
| 2806.000 | Branch2 | camera3 | camera_blurred | 2 | Office | false |
| 3366.000 | Branch1 | doorlock3 | door_battery_low | 1 | Vault | false |
| 3905.000 | Branch2 | camera3 | camera_noise_detected | 1 | Office | false |
| ... | ... | ... | ... | ... | ... | ... |

Table 7.1: Excerpt from `light.csv`, showing the fault sequence used for testing.

### 7.1.1 Sequences for testing

We created three test sequences to assess the system as the load increased. Every sequence is saved as a CSV file in the `sequences/` directory and outlines a list of fault events along with their time offsets (please see Table 7.1 for a sample). The parameters are summarized in Table 7.2.

| Sequence | Events | Duration (s) | Avg. interarrival (s) | MPC triggers |
|---|---|---|---|---|
| Light | 20 | 8 059 | 382.1 | 1 |
| Medium | 400 | 10 772 | 27.0 | 3 |
| Heavy | 2 000 | 10 799 | 5.4 | 8 |

Table 7.2: Characteristics of the three test sequences.

All sequences were created within the following time frame: $\approx$ 3 hours for medium and heavy events, and $\approx$ 2.2 hours for light events, though they varied in terms of event density. The light sequence features infrequent events, with an average interval of 382 seconds between them. Compared to this, the medium sequence presents a moderate density of events taking place every 27 seconds. Finally, the heavy sequence puts significant demands on the system, characterized by frequent events taking place every 5 seconds. The calculation requests for the MPC are automatically coupled based on the sequence parameters, resulting in 1, 3, and 8 triggers, respectively. These sequences enable us to see how latency, throughput, resource usage, and power draw change as the event load increases.

## 7.2 Performance metrics

This analysis highlights the trade-offs between performance and privacy. This shows the impact of blockchain and MPC on factors such as resource utilization and energy consumption, and lays the groundwork for the conversation in Chapter 8. We connect each outcome to the research questions.

### 7.2.1 Overall results

| run | cpu_seconds_total | max_mem_mb | avg_mem_mb | disk_rw_mb | energy_wh | avg_power_w | diff_from_idle_w | blockchain_tx | mpc_calls | avg_latency_ms | duration_h |
|---|---|---|---|---|---|---|---|---|---|---|---|
| heavy_fullsystem_20250614_133851 | 399.86 | 3313.44 | 2242.24 | 212.69 | 324.68 | 108.25 | 8.60 | 2011 | 8 | 13.85 | 3.00 |
| heavy_none_20250614_225137 | 54.31 | 76.29 | 65.88 | 0.55 | 313.85 | 104.64 | 5.00 | 0 | 0 | 5.58 | 3.00 |
| heavy_onlybc_20250614_194719 | 332.53 | 2155.19 | 1874.51 | 128.17 | 320.98 | 106.99 | 7.35 | 2003 | 0 | 15.26 | 3.00 |
| heavy_onlympc_20250614_164313 | 113.69 | 840.30 | 212.00 | 85.18 | 319.36 | 106.49 | 6.84 | 0 | 8 | 3.84 | 3.00 |
| light_fullsystem_20250615_015539 | 181.47 | 2025.06 | 1687.36 | 182.57 | 236.16 | 105.37 | 5.73 | 24 | 1 | 18.26 | 2.24 |
| light_none_20250615_085134 | 1.90 | 57.21 | 55.93 | 0.55 | 230.05 | 102.78 | 3.14 | 0 | 0 | 14.78 | 2.24 |
| light_onlybc_20250615_063303 | 169.78 | 1831.77 | 1609.51 | 97.50 | 235.70 | 105.20 | 5.55 | 23 | 0 | 32.57 | 2.24 |
| light_onlympc_20250615_041430 | 12.27 | 233.39 | 166.14 | 85.18 | 232.59 | 103.93 | 4.28 | 0 | 1 | 4.22 | 2.24 |
| medium_fullsystem_20250615_110955 | 272.22 | 2260.93 | 1931.01 | 206.63 | 316.39 | 105.64 | 5.99 | 406 | 3 | 15.34 | 3.00 |
| medium_none_20250615_202117 | 5.69 | 59.89 | 57.31 | 0.55 | 304.85 | 101.90 | 2.26 | 0 | 0 | 6.68 | 2.99 |
| medium_onlybc_20250615_171731 | 244.44 | 2126.00 | 1792.19 | 121.44 | 315.17 | 105.21 | 5.57 | 403 | 0 | 17.96 | 3.00 |
| medium_onlympc_20250615_141346 | 29.04 | 244.63 | 212.73 | 85.18 | 311.22 | 104.02 | 4.38 | 0 | 3 | 3.11 | 2.99 |

Table 7.3: Resource usage and performance across the 12 controlled runs (Light/Medium/Heavy × None/BC-only/MPC-only/Full). Each row corresponds to one end-to-end run.

| run | container_name | cpu_seconds_total | max_mem_mb | avg_mem_mb | disk_rw_mb |
|---|---|---|---|---|---|
| heavy_fullsystem_20250614_133851 | blockchain_node_0 | 94.21 | 762.58 | 637.39 | 43.23 |
| heavy_fullsystem_20250614_133851 | blockchain_node_1 | 84.04 | 739.28 | 629.03 | 41.70 |
| heavy_fullsystem_20250614_133851 | blockchain_node_2 | 84.10 | 727.62 | 600.88 | 41.57 |
| heavy_fullsystem_20250614_133851 | blockchain_web_interface | 14.80 | 111.23 | 110.29 | 1.01 |
| heavy_fullsystem_20250614_133851 | coordinator | 56.16 | 60.62 | 48.73 | 0.31 |
| heavy_fullsystem_20250614_133851 | iot-simulator | 1.43 | 35.39 | 35.32 | 0.24 |
| heavy_fullsystem_20250614_133851 | mpc_node_0 | 21.83 | 686.90 | 62.88 | 28.21 |
| heavy_fullsystem_20250614_133851 | mpc_node_1 | 21.70 | 943.59 | 60.17 | 28.21 |
| heavy_fullsystem_20250614_133851 | mpc_node_2 | 21.58 | 69.50 | 57.54 | 28.21 |
| heavy_fullsystem_20250614_133851 | TOTAL | 399.86 | 3313.44 | 2242.24 | 212.69 |
| heavy_none_20250614_225137 | coordinator | 52.83 | 49.62 | 39.21 | 0.31 |
| heavy_none_20250614_225137 | iot-simulator | 1.48 | 26.67 | 26.67 | 0.24 |
| heavy_none_20250614_225137 | TOTAL | 54.31 | 76.29 | 65.88 | 0.55 |
| heavy_onlybc_20250614_194719 | blockchain_node_0 | 92.12 | 720.76 | 606.99 | 42.49 |
| heavy_onlybc_20250614_194719 | blockchain_node_1 | 84.85 | 688.76 | 568.81 | 42.00 |
| heavy_onlybc_20250614_194719 | blockchain_node_2 | 84.24 | 700.20 | 578.26 | 42.12 |
| heavy_onlybc_20250614_194719 | blockchain_web_interface | 14.47 | 55.60 | 54.76 | 1.01 |
| heavy_onlybc_20250614_194719 | coordinator | 55.39 | 50.54 | 39.88 | 0.31 |
| heavy_onlybc_20250614_194719 | iot-simulator | 1.45 | 25.82 | 25.82 | 0.24 |
| heavy_onlybc_20250614_194719 | TOTAL | 332.53 | 2155.19 | 1874.51 | 128.17 |
| heavy_onlympc_20250614_164313 | coordinator | 53.02 | 49.34 | 39.77 | 0.31 |
| heavy_onlympc_20250614_164313 | iot-simulator | 1.46 | 25.85 | 25.85 | 0.24 |
| heavy_onlympc_20250614_164313 | mpc_node_0 | 19.88 | 658.71 | 51.13 | 28.21 |
| heavy_onlympc_20250614_164313 | mpc_node_1 | 19.77 | 95.89 | 47.65 | 28.21 |
| heavy_onlympc_20250614_164313 | mpc_node_2 | 19.55 | 94.19 | 47.60 | 28.21 |
| heavy_onlympc_20250614_164313 | TOTAL | 113.69 | 840.30 | 212.00 | 85.18 |

Table 7.4: Per-container metrics per run: CPU, memory and disk. Corresponding energy usage details and the total system latency can be found in Table 7.3 (1/3).

| run | container_name | cpu_seconds_total | max_mem_mb | avg_mem_mb | disk_rw_mb |
|---|---|---|---|---|---|
| light_fullsystem_20250615_015539 | blockchain_node_0 | 57.67 | 633.82 | 505.51 | 32.05 |
| light_fullsystem_20250615_015539 | blockchain_node_1 | 56.51 | 611.73 | 484.34 | 32.29 |
| light_fullsystem_20250615_015539 | blockchain_node_2 | 55.29 | 495.34 | 477.92 | 32.04 |
| light_fullsystem_20250615_015539 | blockchain_web_interface | 0.84 | 54.48 | 54.22 | 1.01 |
| light_fullsystem_20250615_015539 | coordinator | 0.81 | 29.91 | 29.64 | 0.31 |
| light_fullsystem_20250615_015539 | iot-simulator | 1.05 | 25.86 | 25.85 | 0.24 |
| light_fullsystem_20250615_015539 | mpc_node_0 | 3.10 | 60.77 | 37.62 | 28.21 |
| light_fullsystem_20250615_015539 | mpc_node_1 | 3.15 | 57.93 | 36.30 | 28.21 |
| light_fullsystem_20250615_015539 | mpc_node_2 | 3.06 | 58.21 | 35.95 | 28.21 |
| light_fullsystem_20250615_015539 | TOTAL | 181.47 | 2025.06 | 1687.36 | 182.57 |
| light_none_20250615_085134 | coordinator | 0.80 | 31.36 | 30.08 | 0.31 |
| light_none_20250615_085134 | iot-simulator | 1.10 | 25.85 | 25.85 | 0.24 |
| light_none_20250615_085134 | TOTAL | 1.90 | 57.21 | 55.93 | 0.55 |
| light_onlybc_20250615_063303 | blockchain_node_0 | 57.40 | 664.32 | 531.84 | 32.23 |
| light_onlybc_20250615_063303 | blockchain_node_1 | 55.35 | 564.15 | 493.34 | 31.78 |
| light_onlybc_20250615_063303 | blockchain_node_2 | 54.33 | 492.46 | 474.01 | 31.93 |
| light_onlybc_20250615_063303 | blockchain_web_interface | 0.84 | 54.23 | 54.16 | 1.01 |
| light_onlybc_20250615_063303 | coordinator | 0.82 | 30.90 | 29.99 | 0.31 |
| light_onlybc_20250615_063303 | iot-simulator | 1.03 | 26.20 | 26.16 | 0.24 |
| light_onlybc_20250615_063303 | TOTAL | 169.78 | 1831.77 | 1609.51 | 97.50 |
| light_onlympc_20250615_041430 | coordinator | 0.78 | 30.95 | 30.16 | 0.31 |
| light_onlympc_20250615_041430 | iot-simulator | 1.08 | 25.86 | 25.86 | 0.24 |
| light_onlympc_20250615_041430 | mpc_node_0 | 3.46 | 61.21 | 37.79 | 28.21 |
| light_onlympc_20250615_041430 | mpc_node_1 | 3.46 | 58.11 | 36.61 | 28.21 |
| light_onlympc_20250615_041430 | mpc_node_2 | 3.49 | 57.79 | 35.72 | 28.21 |
| light_onlympc_20250615_041430 | TOTAL | 12.27 | 233.39 | 166.14 | 85.18 |

Table 7.4: Per-container metrics per run: CPU, memory and disk. Corresponding energy usage details and the total system latency can be found in Table 7.3 (2/3).

| run | container_name | cpu_seconds_total | max_mem_mb | avg_mem_mb | disk_rw_mb |
|---|---|---|---|---|---|
| medium_fullsystem_20250615_110955 | blockchain_node_0 | 80.81 | 693.44 | 568.85 | 40.70 |
| medium_fullsystem_20250615_110955 | blockchain_node_1 | 78.56 | 671.66 | 554.41 | 40.28 |
| medium_fullsystem_20250615_110955 | blockchain_node_2 | 77.55 | 670.36 | 540.40 | 39.46 |
| medium_fullsystem_20250615_110955 | blockchain_web_interface | 3.77 | 55.01 | 54.41 | 1.01 |
| medium_fullsystem_20250615_110955 | coordinator | 5.00 | 35.30 | 32.62 | 0.31 |
| medium_fullsystem_20250615_110955 | iot-simulator | 1.42 | 25.88 | 25.88 | 0.24 |
| medium_fullsystem_20250615_110955 | mpc_node_0 | 8.53 | 60.55 | 52.48 | 28.21 |
| medium_fullsystem_20250615_110955 | mpc_node_1 | 8.31 | 58.52 | 50.83 | 28.21 |
| medium_fullsystem_20250615_110955 | mpc_node_2 | 8.27 | 58.82 | 51.12 | 28.21 |
| medium_fullsystem_20250615_110955 | TOTAL | 272.22 | 2260.93 | 1931.01 | 206.63 |
| medium_none_20250615_202117 | coordinator | 4.25 | 34.05 | 31.47 | 0.31 |
| medium_none_20250615_202117 | iot-simulator | 1.44 | 25.84 | 25.84 | 0.24 |
| medium_none_20250615_202117 | TOTAL | 5.69 | 59.89 | 57.31 | 0.55 |
| medium_onlybc_20250615_171731 | blockchain_node_0 | 80.05 | 730.79 | 600.01 | 40.40 |
| medium_onlybc_20250615_171731 | blockchain_node_1 | 76.81 | 668.75 | 535.76 | 39.64 |
| medium_onlybc_20250615_171731 | blockchain_node_2 | 77.83 | 685.42 | 544.26 | 39.84 |
| medium_onlybc_20250615_171731 | blockchain_web_interface | 3.67 | 54.96 | 54.36 | 1.01 |
| medium_onlybc_20250615_171731 | coordinator | 4.71 | 33.57 | 31.39 | 0.31 |
| medium_onlybc_20250615_171731 | iot-simulator | 1.37 | 26.46 | 26.41 | 0.24 |
| medium_onlybc_20250615_171731 | TOTAL | 244.44 | 2126.00 | 1792.19 | 121.44 |
| medium_onlympc_20250615_141346 | coordinator | 4.36 | 38.16 | 31.66 | 0.31 |
| medium_onlympc_20250615_141346 | iot-simulator | 1.41 | 25.86 | 25.86 | 0.24 |
| medium_onlympc_20250615_141346 | mpc_node_0 | 7.82 | 64.91 | 53.43 | 28.21 |
| medium_onlympc_20250615_141346 | mpc_node_1 | 7.71 | 58.49 | 50.92 | 28.21 |
| medium_onlympc_20250615_141346 | mpc_node_2 | 7.74 | 58.42 | 50.86 | 28.21 |
| medium_onlympc_20250615_141346 | TOTAL | 29.04 | 244.63 | 212.73 | 85.18 |

Table 7.4: Per-container metrics per run: CPU, memory and disk. Corresponding energy usage details and the total system latency can be found in Table 7.3 (3/3).

### 7.2.2 Column definitions and computation

- `cpu_seconds_total` - For every container, gather its cumulative `cpu_seconds` series from the Docker-stats CSV. In the TOTAL row, we sum up the values of all the containers.
- `max_mem_mb` - For each container, we read the `mem_mb` values and determine the maximum value, which is represented by `max_mem_mb`. In the TOTAL row, we sum the `mem_mb` values of all containers, and then determine the overall maximum from those sums.
- `avg_mem_mb` - For each container, we calculate the average of its `mem_mb` series. For the TOTAL row, we sum the `mem_mb` values of all containers and then calculate the average of those sums.
- `disk_rw_mb` - For every container, we read the `size_rw_mb` values. This refers to the R/W layer of the container (the added data by our services) and determine the maximum. In the TOTAL row, we sum the `size_rw_mb` of all containers.
- `energy_wh` - We calculate the duration of each sequence in seconds. First, we calculate the product of power and duration. After that, take the result and divide it by 3600.
- `avg_power_w` - To calculate this, we take `energy_wh`, multiply it by 3600, and then divide the result by the total run duration in seconds. This results in the average power consumption measured in watts.
- `diff_from_idle_w` - We calculate the difference between the average power in watts (`avg_power_w`) and the average baseline idle power obtained from `baseline_energy.csv`. The file contains the average idle system power draw recorded over a one-hour period.
- `blockchain_tx` - We count the number of rows present in the blockchain report CSV files. Every row corresponds to a single on-chain transaction.
- `mpc_calls` - In the MPC report CSV (`mpc_report_.csv`), we count the number of rows where `triggered` is true.
- `avg_latency_ms` - To calculate this, we use the coordinator report CSV file (`report_.csv`) and compute the average of the values in the `ms_total_lifecycle` column.
- `duration_h` - We calculate the run duration by examining the first and last timestamps. To convert to hours, we subtract the values and then divide the result by 3600.

### 7.2.3 Overall results

The summary in Table 7.3 shows an overview of CPU time, memory usage, disk usage, energy consumption, average power draw, on-chain transactions, MPC calls, and end-to-end latency for all twelve experimental runs. In the baseline scenario, where neither blockchain and MPC is used, the system operates with less than 55 CPU-seconds, consumes under 80 MB of RAM, and generates less than 1 MB of disk traffic, while maintaining an average latency of below 6.7 ms.

Table 7.4 shows metrics categorized by container for each run, illustrating the impact of each service component (the blockchain nodes, MPC nodes, coordinator, and simulator) to the overall figures for CPU, memory and disk[1].

The simultaneous use of both blockchain and MPC increases CPU load, ranging from three to seven times higher than the system that just saves the log (baseline with all systems disabled). The memory usage can grow from ten to a hundred times greater, and disk usage can reach between 120 and 215 MB. The average power draw increases by 5 to 9 watts compared to baseline. The latency increases to a range of 13 to 18 milliseconds.

### 7.2.4 MPC overhead

We assessed the cost associated with each three-party SPDZ invocation by examining computation time, communication volume, and the number of protocol rounds involved. The SPDZ-specific results are consistent between invocations, and are shown in Figure 7.1. The output seen is the verbose output from the spdz binaries, and can be accessed from within the Docker desktop application.

[1] Energy usage details and the total system latency is only measured as a system-total per run, and therefore not present in the per-service tables.

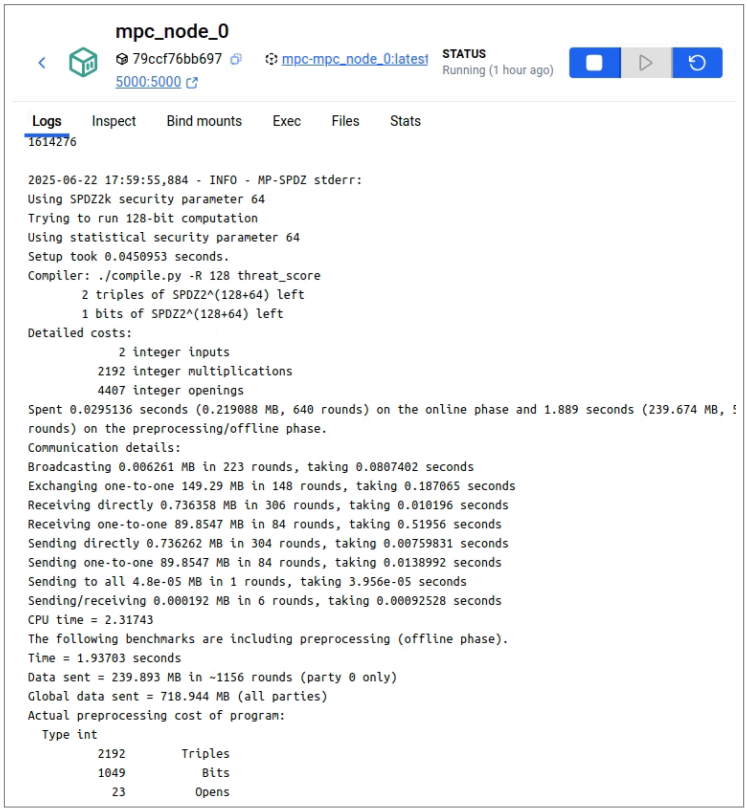



Figure 7.1: Output from MPC node 0 upon triggering an MPC calculation.

- **Computation time per invocation:**
  - ≈2.3 s per party
- **Network traffic per invocation:**
  - ≈239.9 MB sent by each party (total ≈718.9 MB across all three parties)
  - Breakdown per party:
    - Broadcasts: 0.006 MB over 223 rounds
    - One-to-one exchanges: 149.3 MB sent in 148 rounds, 89.9 MB received in 84 rounds
    - Direct peer-to-peer: 0.736 MB sent in 304 rounds, 0.736 MB received in 306 rounds
    - Control messages (small opens, acks): 0.00019 MB in 6 rounds
- **Protocol rounds:**
  - Total ≈1 156 synchronization rounds per invocation
  - Offline (preprocessing) ∼516 rounds, online ∼640 rounds

### 7.2.5 Full system - heavy setting

This scenario tested the system under full load. We tested the system with 2000 events in 3 hours with 8 MPC triggers ("heavy_fullsystem", as defined in Table 7.3):

- **CPU**: 400 s total vs. 54 s baseline.
- **Memory**: 3.3 GB peak, 2.2 GB average vs. 76 MB peak baseline.
- **Disk usage**: 213 MB vs. 0.55 MB baseline.
- **Power**: 108.3 W avg. draw (+8.6 W from baseline).
- **Latency**: 13.9 ms avg. vs. 5.6 ms avg. baseline.

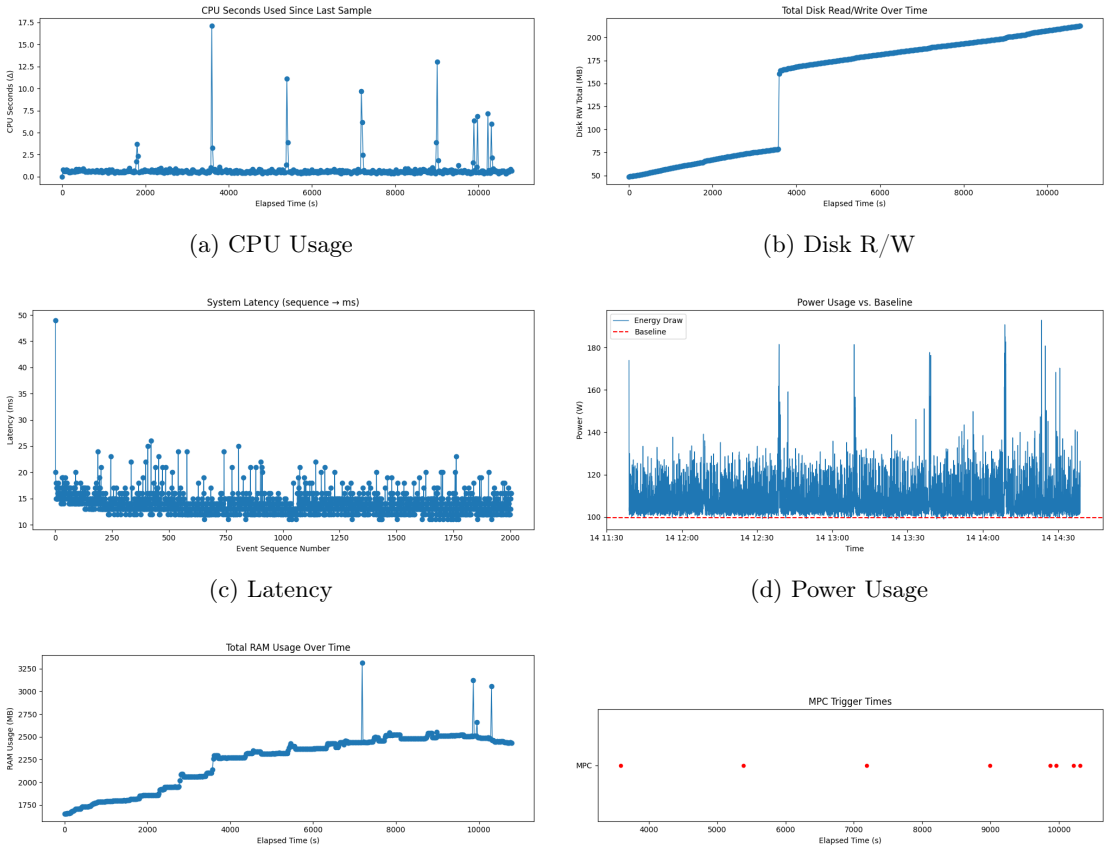
Breakdown per service:

- *Blockchain nodes (x3)* consumed 262 s CPU, handled 2 011 transactions, and added ≈ 15 ms per write.
- *MPC nodes (x3)* consumed 65 s CPU, handled 8 invocations.
- *Coordinator + Simulator* remained lightweight (≈ 58 s CPU, < 60 MB RAM).

The system metrics under this load level can be seen in Figure 7.2.

In panel (a), we observe that CPU usage remains close to one second per sample, except when there is an MPC calculation, then we can see a noticeable spike in CPU seconds, reaching a peak of ≈ 17 seconds.

Panel (b) illustrates the total disk storage usage. We can see an initial spike at startup, followed by a consistent linear increase. In panel (c), the end-to-end latency generally remains within the range of 10 ms to 20 ms for the majority of events, with a peak at 50 ms during startup. Panel (d) illustrates the power draw, showing that the system can reach a peak of around 180 W during processing bursts. Panel (e) illustrates the RAM usage, starting at 1.7 GB and stabilizing around 2.5 GB, with each trigger resulting in a slight increase. Lastly, panel (f) highlights the eight MPC trigger times, with each red dot corresponding to spikes noted in CPU, latency, and RAM.




(a) CPU Usage

(b) Disk R/W

(c) Latency

(d) Power Usage

(e) RAM Usage

(f) Triggers

Figure 7.2: System metrics under heavy load

### 7.2.6 Full system - medium setting

Situation: 400 events over 3 h and 3 MPC triggers ("medium_fullsystem", as defined in Table 7.3):

- **CPU**: 272 s vs. 5.7 s baseline.
- **Memory**: 2.26 GB peak, 1.93 GB avg. vs. 59 MB baseline.
- **Disk usage**: 207 MB vs. 0.55 MB baseline.

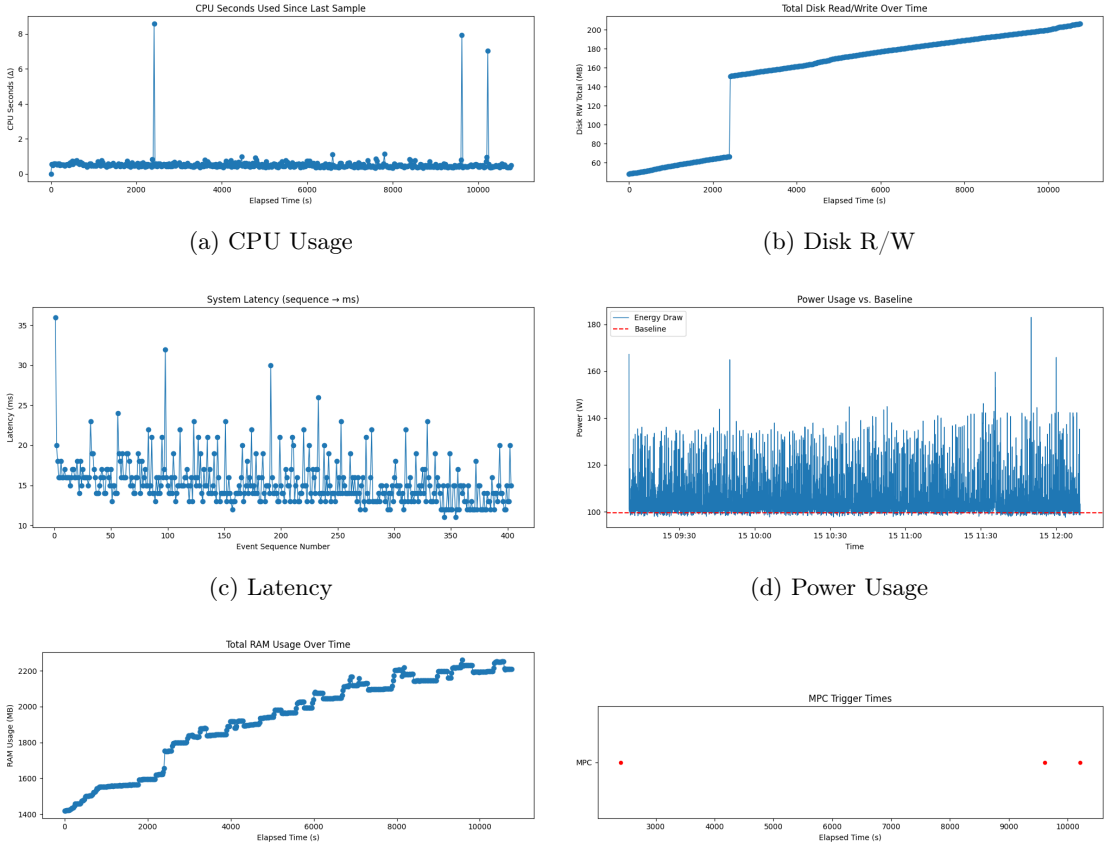
- **Power**: 105.6 W (+5.9 W from baseline).
- **Latency**: 15.3 ms avg. vs. 6.7 ms avg. baseline.

Breakdown per service:

- *Blockchain*: Adds 237 s CPU, performs 406 writes, and adds ≈ 18 ms per write.
- *MPC*: Adds 25,1 s CPU, 3 calls.

The system metrics under this load level can be seen in Figure 7.3.

In panel (a), the CPU remains close to one second per sample. Each MPC trigger generates spikes lasting up to 8 CPU seconds. In panel (b), we can observe the surge at startup, which is subsequently accompanied by a steady rise in total disk usage, reaching approximately 200 MB. Panel (c) shows that latency is primarily within the 15-18 ms range, although there are occasional spikes that exceed 30 ms. Panel (d) shows that the power draw can have peaks reaching around 175 W. The RAM usage in panel (e) increases from 1.5 GB to approximately 2.2 GB, showing slight upward fluctuations. Panel (f) highlights the three MPC calls, each corresponding with noticeable spikes in CPU, memory, and latency.




(a) CPU Usage

(b) Disk R/W

(c) Latency

(d) Power Usage

(e) RAM Usage

(f) Triggers

Figure 7.3: System metrics under medium load

### 7.2.7 Full system - light setting

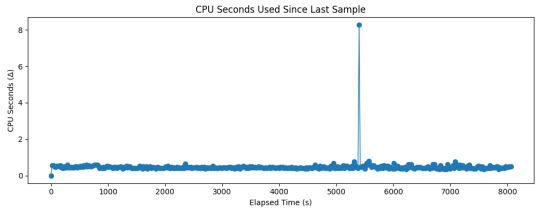
Situation: 20 events in 2.2 h with 1 MPC trigger ("light_fullsystem", as defined in Table 7.3):

- **CPU**: 181 s vs. 1.9 s baseline.
- **Memory**: 2.03 GB peak, 1.69 GB avg. vs. 57 MB baseline.
- **Disk usage**: 183 MB vs. 0.55 MB baseline.
- **Power**: 105.4 W (+5.7 W from baseline).

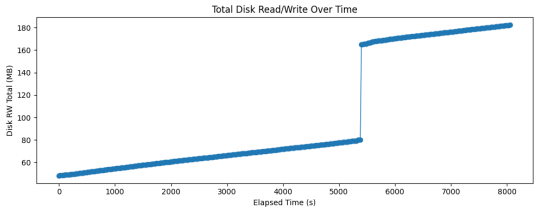
- **Latency**: 18.3 ms avg. vs. 14.8 ms avg. baseline.

Here, block sealing frequency dominates latency (32.6 ms per write) since writes occur infrequently. The single MPC call added 9.3 CPU seconds. The system metrics under this load level can be seen in Figure 7.4.

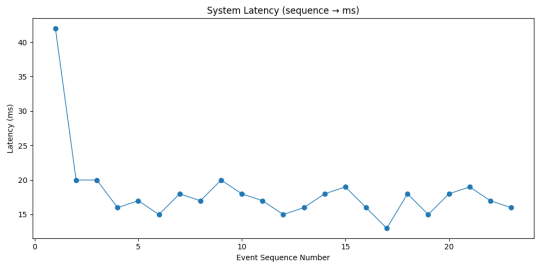
Figure 7.4 shows that in panel (a), we observe that the CPU remains close to one second per sample. One spike occurs during the single MPC call, when it reaches approximately 8 seconds. Panel (b) shows a substantial increase in disk usage, occurring when the MPC system is first triggered, after which the activity levels off, which seems to be consistent behaviour between runs. The latency observed in panel (c) is approximately 15 ms, featuring two notable peaks around 40 ms during startup. In panel (d), we can observe that the power draw can reach up to 180 W, when the MPC call is requested. In panel (e), we observe that RAM usage gradually increases from 1.4 GB to approximately 2.0 GB, with a notable spike occurring during the MPC call. Panel (f) indicates the only MPC trigger, which corresponds with the spikes.



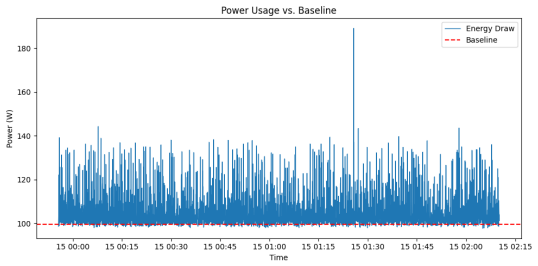

(a) CPU Usage



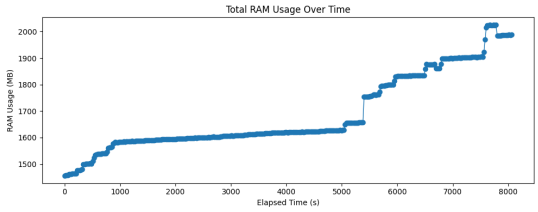

(b) Disk R/W



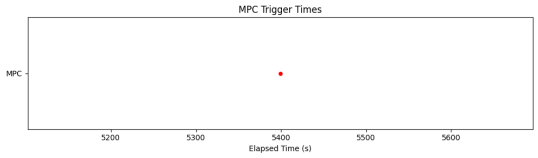

(c) Latency




(d) Power Usage




(e) RAM Usage




(f) Triggers

Figure 7.4: System metrics under light load

### 7.2.8 Component-only configurations

To isolate each technology's cost, we ran two partial configurations.

#### 1: Blockchain only

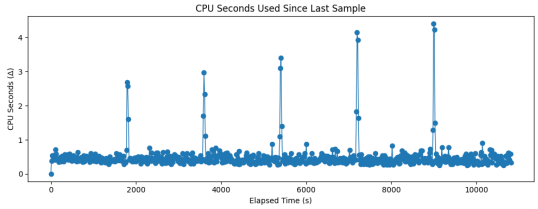
Disabling MPC (defined as "onlybc_*" in Table 7.4), in the heavy setting, results in:

- CPU and memory nearly match the heavy full system test (333 s vs. 400 s CPU under heavy load).
- Disk usage ≈ 60-65% of full-system (e.g. 128 MB vs. 213 MB).
- Power draw +6-7 W above baseline.

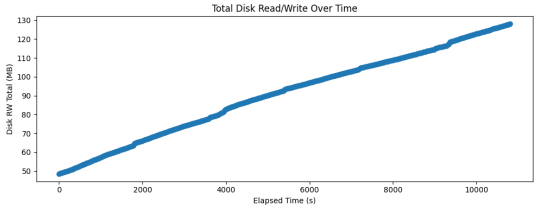
- Latency ≈ 15-18 ms.

Blockchain alone accounts for the majority of the resource and latency overhead. The system metrics can be seen in Figure 7.5.

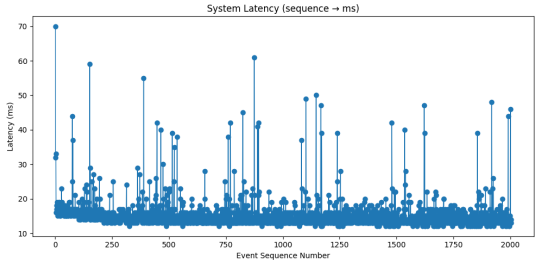
This figure shows the performance impact of the isolated blockchain component during the heavy sequence. In panel (a), we observe CPU spikes reaching up to 4 seconds per sample, while the baseline usage remains close to zero. Panel (b) illustrates a consistent increase in disk size, reaching approximately 130 MB. The latency values shown in panel (c) stay under 15 ms for almost all write operations, but can have peaks reaching around 60 ms. Panel (d) shows that peaks can reach up to 180 W. The RAM usage in panel (e) peaks around 2100 MB for the full system.



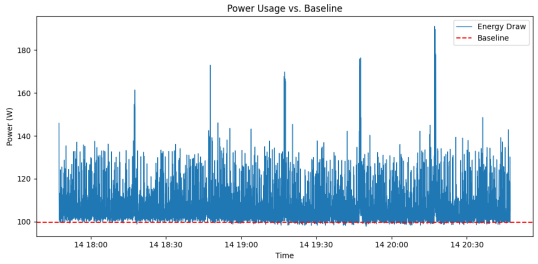

(a) CPU Usage



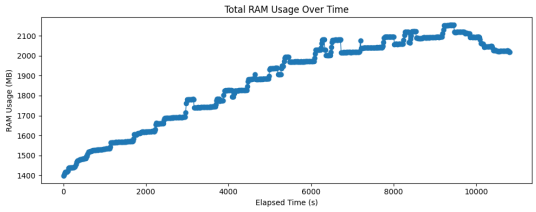

(b) Disk R/W




(c) Latency




(d) Power Usage




(e) RAM Usage

Figure 7.5: Blockchain-only system metrics under heavy load

**2: MPC only**

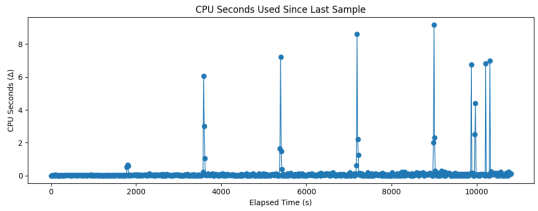
Disabling blockchain (defined as "onlympc_*" in Table 7.4 ), results in:

- CPU: 114 s (heavy), 29 s (medium), 12 s (light).
- Memory: peaking at 840 MB, with 212 MB on average (heavy), 245 MB peak, 213 MB average (medium), and 234 MB peak, 166 MB average (light).
- Disk usage: ≈ 85 MB.
- Power: +4-7 W.

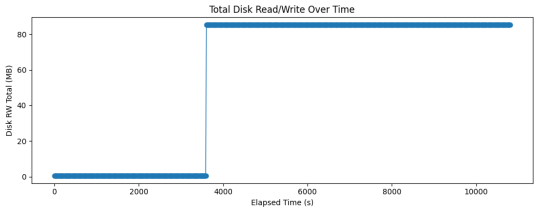
MPC adds modest, predictable overhead when it is not triggered too regularly. The system metrics can be seen in Figure 7.6.

Figure 7.6 examines the MPC service while it is experiencing the same heavy load as in the blockchain test. In panel (a), we can see that CPU usage experiences spikes around 8 seconds during each of the eight MPC invocations, whereas the idle CPU remains close to zero. In panel (b), we can see that disk

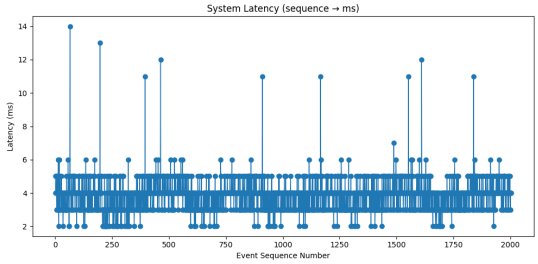
usage increases to approximately 80 MB when the first trigger is requested. Panel (c) illustrates that latency remains under 6 ms for routine logging, although there are occasional spikes to 14 ms. The power draw in panel (d) can reach a peak of approximately 185 W. The RAM usage in panel (e) remains below 300 MB, except for a single spike reaching 800 MB. Panel (f) highlights the eight MPC trigger times, which correspond with the CPU and power spikes.



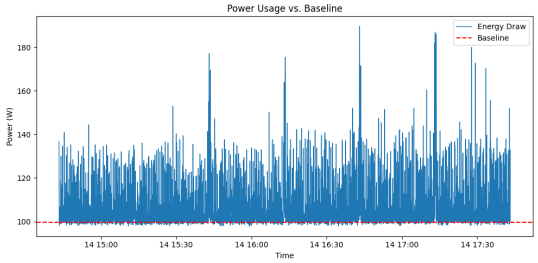

(a) CPU Usage



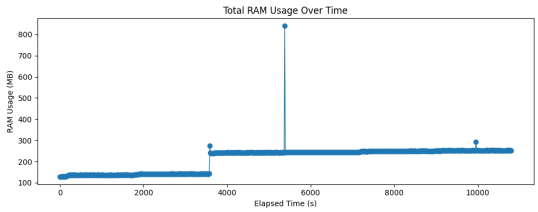

(b) Disk R/W



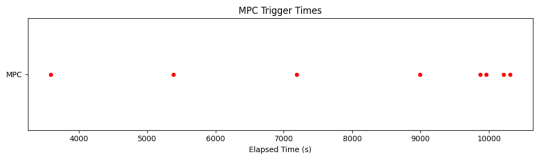

(c) Latency




(d) Power Usage




(e) RAM Usage



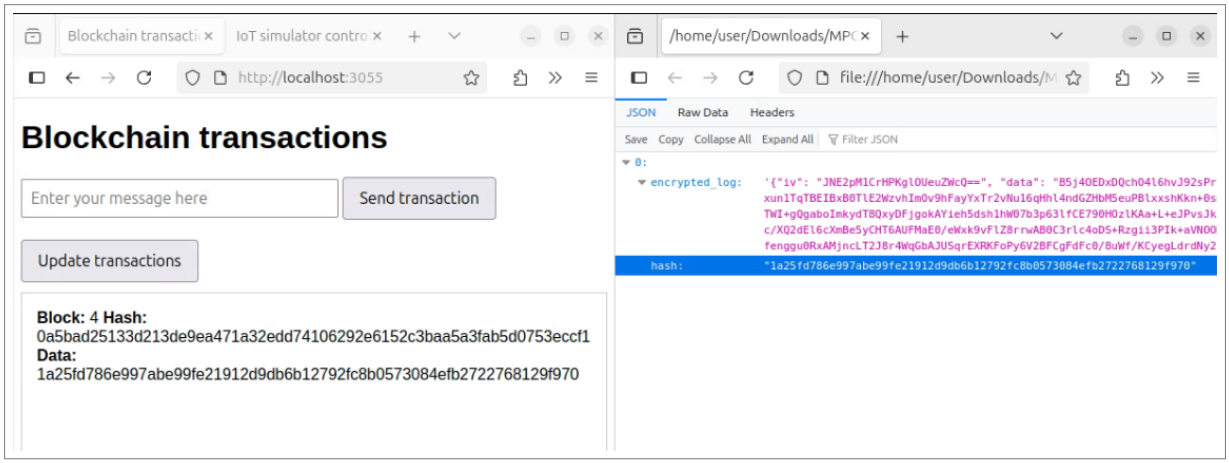

(f) Triggers

Figure 7.6: MPC-only system metrics under heavy load

## 7.3 Hybrid on-chain / off-chain storage system

We find that the prototype guarantees that every fault entry is encrypted and stored locally by the coordinator. The coordinator calculates the SHA-256 hash of this data and only publishes that hash on the blockchain, as illustrated in Figure 7.7. In the encrypted view, the off-chain log shows only the AES-CBC ciphertext. In the decrypted view, the same fields are shown in clear text for demonstration purposes. In both instances, the blockchain display features only two hashes: the hash of the block itself and the SHA-256 fingerprint of the event. Since no actual fault data is ever part of the chain, anyone accessing the blockchain data gains no insight into the security events that take place.

At the same time, if the data needs to be verified, the off-chain log can be accessed and compared against the on-chain data containing the hash to prove its integrity. When a log needs to be erased, deleting the local encrypted file does not change any data that is stored on-chain, but the data will still be erased. This combination results in reliable audit trails that cannot be modified, but it also keeps sensitive information private while also making it able to be deleted.



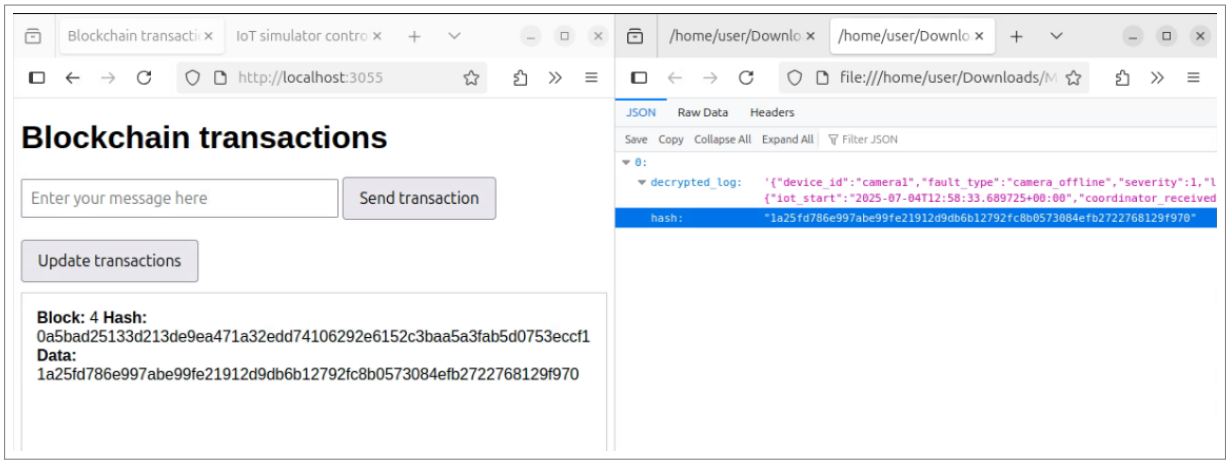




Figure 7.7: Coordinator's local event log, first shown encrypted (top) and then decrypted for illustration purposes (bottom), alongside the corresponding blockchain data. In both cases only the SHA-256 hash of each log entry is published on chain. In the blockchain view you see two hashes: the first is the block's hash, and the second is the hash of the logged fault.

# Chapter 8

# Discussion

This chapter discusses the findings and offers an in-depth evaluation of how the prototype performs in the aspects of privacy, integrity, and performance. We base the interpretations on the data we gathered, and we suggest design choices based on the evaluations.

## 8.1 Overall prototype performance and privacy properties

Our controlled experiments show that the hybrid on-chain / off-chain logging method stores only cryptographic hashes on the blockchain, while keeping full encrypted logs off-chain. The end-to-end latency remained in the double digit milliseconds even during heavy load, suggesting that the system is capable of processing real-time faults. Measurements of resources indicate that CPU and memory usage increase in a pattern that is predictable as event volume increases, and the noted rise in power consumption seems to be minimal on average.

We examined four different configurations: the baseline (which lacks both blockchain and MPC), blockchain alone, MPC alone, and the complete system. This approach allowed us to identify the cost contributions from each component.

For MPC, we find that implementing a three-party SPDZ$^{2k}$ protocol adds a noticeable computational cost. In the most demanding test scenario, we observed that MPC added about 114 seconds to the total CPU time when compared to a baseline that uses only local logging. On average, each invocation of the MPC contributed just single digit milliseconds to the overall latency [1], while the maximum memory usage increased by 840 megabytes. The increase in power usage ranged from 4 to 7 watts on average.

In terms of network activity, each party transmits and receives approximately 240 MB of data with each invocation, leading to a total of around 720 MB when looking at all three parties involved. The transfers take place over around 1,156 synchronization rounds, which include 516 rounds for offline preprocessing and 640 rounds for online processing. This indicates a considerable amount of bandwidth usage that increases with the number of MPC triggers.

Overall, in high-security scenarios, the advantages of privacy offered by MPC may outweigh these performance drawbacks. These performance drawbacks can be reduced if we are willing to accept reduced numeric precision and reduced security margins. For instance, using $10^2$ to $10^4$ rather than $10^6$ helps maintain smaller numbers. This can allow for a smaller ring size $k$, which reduces bandwidth and memory.

Also, MPC can be arranged to take place at regular intervals rather than being triggered solely when necessary. In the current design, MPC nodes calculate the global threat score whenever a manual request is made. This process might unintentionally reveal which branch started the computation, and it may be triggered due to a significant fault in that branch. To address this, we can separate the execution of the MPC from event triggers by scheduling the protocol to run at regular intervals, such as every 10 minutes, or by combining several requests into one computation. This modification ensures that no branch can associate a computation request with a particular event or branch, thereby maintaining

[1] Please see limitations, MPC calculations are performed using parallel processing

the confidentiality of the inputs from each individual branch, while also making sure the system is not overwhelmed by requests.

The experiments demonstrate that integrating the blockchain layer results in a consistent resource and latency cost. In the blockchain-only setup under heavy load, the three validator nodes used approximately 333 CPU-seconds, which took up 83% of the total CPU cost for the system under full load in the heavy test setting. The system reached a maximum of 2.1 GB of RAM usage along with around 128 MB of data written to disk over a three-hour period. The average power draw increased by 6 to 7 watts compared to the baseline. The end-to-end latency for each logging event increased to between 15 and 18 milliseconds, with some occasional spikes. The disk usage increased in a linear fashion as the number of transactions rose, averaging around 0.06 MB for each write operation, for all three nodes, which results in 0,02 MB per node.

The blockchain layer introduces overhead, but in return, it provides cryptographic integrity for a distributed log ledger.

### 8.1.1 Effects of event volume and frequency

We conducted tests at three different workload levels (light, medium, and heavy) to evaluate the scalability of the system in relation to event volume and fault frequency, which reveals the following data:

**Light scenario (20 events, 1 MPC trigger)**

- **Baseline** (no blockchain, no MPC): 1.9 s CPU, 57 MB RAM, 0.6 MB disk, 14.8 ms latency. Power difference from idle: 3.14 W.
- **Full system**: 181 s CPU (+179 s), 2.03 GB peak RAM (+1.97 GB), 183 MB disk (+182 MB), 18.3 ms latency (+3.5 ms). The power difference from idle: 5.73 W; +2.59 W over baseline.
- The single MPC trigger added an 8 s CPU.

**Medium scenario (400 events, 3 MPC triggers)**

- **Baseline**: 5.7 s CPU, 59 MB RAM, 0.6 MB disk, 6.7 ms latency. Power difference from idle: 2.26 W.
- **Full system**: 272 s CPU (+266 s), 2.26 GB peak RAM (+2.20 GB), 207 MB disk (+206 MB), 15.3 ms latency (+8.6 ms). The power difference from idle: 5.99 W; +3.73 W over baseline.
- Each of the three MPC triggers caused CPU spikes up to around 8 s.

**Heavy scenario (2000 events, 8 MPC triggers)**

- **Baseline**: 54 s CPU, 76 MB RAM, 0.6 MB disk, 5.6 ms latency. Power difference from idle: 5.00 W.
- **Full system**: 400 s CPU (+346 s), 3.31 GB peak RAM (+3.23 GB), 213 MB disk (+212 MB), 13.9 ms latency (+8.3 ms). The power difference from idle: 8.60 W; +3.60 W over baseline.
- The eight MPC triggers led to repeated CPU spikes of about 8 s each and corresponding memory and power peaks.

In all three settings, overhead from blockchain and MPC grows with event count and trigger frequency. Blockchain alone contributes most of the CPU, memory and disk growth. MPC added short compute and network spikes at each trigger. Latency rises to double-digit milliseconds. Overall, the system can process up to 2000 security events in under three hours, maintain end-to-end latency below 20 ms, and deliver integrity and privacy guarantees with resource usage that scales with workload.

### 8.1.2 Limitations

Although our prototype shows promising results in controlled tests, there are several limitations that should be considered when interpreting these findings.

First, we test the system using synthetic fault events. This approach allows us to have full control over the timing and load of the events. This method does not account for a variety of real-world factors, such as variable network latency, packet loss, and jitter, which are not included in our evaluations.

Also, by running all services on one virtual machine in optimal network conditions, we can avoid the challenges that come with a real-world distributed setup, which can alter the results in a real-world scenario. In a real-world deployment, every branch would also operate its own "orchestrator" node along with a blockchain interface, resulting in three of each in our three-branch scenario. These are currently shared in our current proof of concept. The multi-orchestrator setup can introduce additional inter-service coordination and associated latency.

In our experiments, we focus solely on a minimal three-party MPC setup, which is combined with a corresponding three-validator private blockchain. Changes in the network size or topology can impact communication complexity and costs.

Our prototype shows that using SPDZ$^{2k}$ within the MP-SPDZ framework is feasible, but this performance is closely linked to the specific protocol and framework selected. Exploring alternative protocols as well as different implementations (frameworks) may (significantly) alter the characteristics of CPU, memory, network, and latency. Overall, the results represent just one aspect of the design space. They might not apply broadly if there are changes to the underlying MPC or the blockchain layer.

From a security perspective, we do not expose the system to adversarial challenges. We do not conduct tests involving malicious nodes, network-level attacks, or key compromises. A comprehensive threat analysis is important for identifying practical vulnerabilities and strengthening the system against active attackers.

In terms of hardware, our server which performs the experiments is equipped with an AMD Ryzen 7 5800x processor. Devices that have limited resources, like older hardware, could show a greater overhead in terms of CPU, memory, and power usage. The impact on performance might be more noticeable on less capable servers.

The method we use to gather our data can influence the results we observe during the collection process. We collect Docker statistics every five seconds to prevent overloading the API. However, each polling cycle could take as long as 1.8 seconds to finish, which means that short, small resource spikes might go unnoticed. The measurements of power draw are also influenced by a polling interval, set by the hardware and software we utilize for measuring the power draw (the hub and the power measuring device, and also the Home Assistant server receiving the data).

And lastly, we deliberately execute each MPC computation in a background thread to keep per-event latency low. If we run MPC synchronously (not in the background), the several seconds required for each invocation can significantly affect the overall system latency.

# Chapter 9

# Conclusions

## 9.1 Answers to the research questions

- **SQ1: How does a decentralized fault detection system leveraging blockchain improve the security and integrity of fault logs compared to a centralized alternative?**

  The system logs every fault hash on a private proof-of-authority blockchain. Across branches, the validators jointly seal blocks in a consensus process. This approach helps to secure the data on the blockchain. Modifying off-chain logs changes the hash. Auditors have the ability to verify logs by comparing the hashes stored on-chain with the entries stored off-chain. The distributed ledger provides a level of tamper resistance and ensures that it remains continuously available.

- **SQ2: What performance penalty does multi-party computation introduce in high-security IoT fault detection?**

  Every time a MPC computation request is made, it results in approximately 2.3 seconds of computational time for each party involved (around 7 seconds in total) and generates around 240MB of network traffic for each node. The end to end latency is not significantly impacted when the computation is being run in the background. The memory usage increases by less than 1GB, while the power consumption goes up by 4 to 7 watts. These costs remain consistent per request.

- **SQ3: What are the trade-offs between decentralization, auditability, and regulatory compliance in an IoT-based security monitoring system?**

  By decentralizing fault detection across multiple branches, we can avoid a single point of failure in the IoT systems, such as vault monitoring equipment. When we use a private blockchain, each branch is able to encrypt and store its own security events on-site and only shares cryptographic hashes on the private blockchain ledger. This makes sure that an immutable audit trail is made available. MPC enables different branches to collaboratively calculate a joint threat score aimed at identifying coordinated attacks while keeping the input data confidential.

  The hybrid on-chain / off-chain storage model that combines on-chain and off-chain elements stores only SHA-256 hashes on the blockchain, while it keeps the complete logs encrypted off the chain. This design ensures that auditability is possible while also facilitating compliance with data erasure or modification requirements.

- **SQ4: How do latency, throughput, and resource overhead compare between system configurations with and without blockchain and MPC?**

  We performed tests on four different system configurations: a baseline setup without blockchain or MPC, one with only blockchain, another with only MPC, and at last, the complete system. The goal is to assess latency, throughput, and resource usage across these configurations.

In the heavy test scenario, the baseline configuration (without blockchain or MPC) manages to process 2,000 events while utilizing 54 CPU-seconds, 76 MB of RAM, and 0.6 MB of disk space, resulting in an average latency of 5.6 milliseconds per event. When we introduce just the blockchain component, we see a large rise in CPU consumption, reaching 333 CPU-seconds, a 6.2-fold increase. Additionally, peak memory usage grows to 2.1 GB, marking a 27.6-fold increase, while latency increases to between 15 and 18 milliseconds per event, making it 2.7 to 3.2 times slower.

In comparison, also in the heavy test scenario, the MPC-only setup utilizes 114 CPU-seconds (representing a 2.1-fold increase) and requires 840 MB of RAM (an 11-fold increase). It maintains the per-event latency at the baseline of 5.6 ms, due to parallel processing. MPC adds eight processing events of 2.3 CPU-seconds per party, where each party exchanges also shares about 240 MB of data. Because the MPC calculations do not block the main process, it does not effect the thoughput of the system.

The complete system, which includes both blockchain and MPC, utilizes 400 CPU-seconds results in a 7.4-fold overhead. The system reaches a peak memory usage of 3.3 GB of RAM, which is 43.4 times higher than the baseline, and requires 213 MB of disk space, reflecting a 355-fold increase. Additionally, the system maintains an average latency of 13.9 milliseconds per event.

For a system that combines both blockchain and MPC, this latency is likely acceptable for real-time IoT monitoring, demonstrating that such integration is feasible. The blockchain layer dominates CPU/memory/disk, while MPC adds predictable bursts of CPU, memory and bandwidth each time it runs.

## 9.2 Future work

The first opportunity for future research is to improve the current hybrid on-chain / off-chain storage model by including threshold decryption capabilities. In this system, each log entry would be encrypted using a public key, with the corresponding private key divided between several branches. The decryption key can only be reconstructed when a specific number of branches work together using a MPC protocol. This approach ensures that no single party holds enough information to independently decrypt logs. At the same time, it permits authorized auditors to access and examine specific entries as needed.

Another important area to investigate is how the system responds during (simulated) branch outages, and research how the system can be improved to work better in these scenarios. This research can help identify vulnerabilities and help with enhancements to maintain high availability, even under challenging conditions.

For blockchain specifically, comparing proof-of-authority, proof-of-trusted-work, and proof-of-stake consensus mechanisms in permissioned IoT deployments could present important information which could alter the protocol choice. Measuring CPU, memory, and network utilization for each protocol under the same workloads would allow for identification of which approach effectively balances security, throughput, and resource efficiency the best.

Besides the aforementioned protocols, an overall broader, systematic comparative analysis of all available other consensus methods would help with the selection the most appropriate protocol for this specific application. Investigating performance characteristics, security guarantees, and how well they fit into resource-constrained edge environments would possibly bring forward a more appropriate consensus mechanism.

For MPC specifically, it would be beneficial to conduct a systematic benchmark for different alternative MPC protocols and frameworks, ensuring that they are tested under the same IoT-edge workloads. This should encompass MPC protocols along with different frameworks. To determine the best trade-offs for each threat model, it is important to compare key metrics such as CPU usage, memory consumption, network traffic, round-trip times, and end-to-end latency directly against one another.

To improve the system further, investigating techniques like anomaly scoring to evaluate the significance of individual log entries in a dynamic way could help with real-world usage. For example, by giving each event a “priority” score, the system can more effectively enable the ability to audit the most critical security incidents first, placing the focus on the most important issues.

Additionally, research into the expansion of the number of branches in the network is important to explore how growth affects system metrics. Research how growth in terms of branches affects consensus latency, end-to-end throughput, MPC communication complexity, blockchain storage growth, network bandwidth consumption, and the usage of CPU and memory. A scalability study like this, which involves changing the number of participating nodes, would help identify bottlenecks and advise architectural improvements for larger deployments.

Finally, expanding the system from just one edge server to a network of local edge nodes could allow for improved redundancy and load balancing. Utilizing an offloading framework that intelligently allocates tasks among nodes, while considering factors like energy consumption, round-trip latency, available bandwidth, and data size can mitigate load spikes and ensure a failover method in case of individual node failures. This multi-node edge architecture could have the potential to expand the possibilities in low-latency, privacy-preserving IoT fault detection.

# Bibliography


[1] R. Hasan. Design framework for internet of things based next generation video surveillance. Master's thesis, University of Saskatchewan, 2017. URL https://harvest.usask.ca/bitstream/10388/8320/1/HASAN-THESIS-2017.pdf.

[2] S. Rizvi, R. Pipetti, N. McIntyre, J. Todd, and I. Williams. Threat model for securing internet of things (iot) network at device-level. *Internet of Things*, 9:100073, 2020. URL https://www.sciencedirect.com/science/article/pii/S2542660520300731.

[3] Md Ashraf Uddin, Andrew Stranieri, Iqbal Gondal, and Venki Balasubramanian. A survey on the adoption of blockchain in IoT: challenges and solutions. *Blockchain: Research and Applications*, 2 (2):100006, 2021. ISSN 2096-7209. doi: https://doi.org/10.1016/j.bcra.2021.100006. URL https://www.sciencedirect.com/science/article/pii/S2096720921000014.

[4] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. 06 2017. doi: 10.1109/BigDataCongress.2017.85. URL https://www.researchgate.net/publication/318131748_An_Overview_of_Blockchain_Technology_Architecture_Consensus_and_Future_Trends.

[5] European Parliament and Council of the European Union. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation), may 2016. URL https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng.

[6] Wenxian Li, Yong Feng, Nianbo Liu, Yingna Li, Xiaodong Fu, and YongTao Yu. A secure and efficient log storage and query framework based on blockchain. *Computer Networks*, 252:110683, 2024. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2024.110683. URL https://www.sciencedirect.com/science/article/pii/S1389128624005152.

[7] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu an Tan. Secure Multi-Party Computation: Theory, practice and applications. *Information Sciences*, 476:357–372, 2019. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2018.10.024. URL https://www.sciencedirect.com/science/article/pii/S0020025518308338.

[8] Dengzhi Liu, Geng Yu, Zhaoman Zhong, and Yuanzhao Song. Secure multi-party computation with secret sharing for real-time data aggregation in IIoT. *Computer Communications*, 224:159–168, 2024. ISSN 0140-3664. doi: https://doi.org/10.1016/j.comcom.2024.06.002. URL https://www.sciencedirect.com/science/article/pii/S0140366424002068.

[9] Shivani Wadhwa, Shalli Rani, Kavita, Sahil Verma, Jana Shafi, and Marcin Wozniak. Energy Efficient Consensus Approach of Blockchain for IoT Networks with Edge Computing. *MDPI Sensors*, 22, 10 2022. URL https://www.mdpi.com/1424-8220/22/10/3733.

[10] Sheetal Zalte-Gaikwad. Edge Computing Technology: An Overview. *ResearchGate*, 7:96–99, 03 2022. URL https://www.researchgate.net/publication/359616603_Edge_Computing_Technology_An_Overview.

[11] M Asif, S Wang, MF Shahzad, and M Ashfaq. Data privacy and cybersecurity challenges in the digital transformation of the banking sector. *Computers & Security*, 2024. URL https://www.sciencedirect.com/science/article/pii/S0167404824003560.

[12] Antar Shaddad Abdul-Qawy, PJ Pramod, E Magesh, and T Srinivasulu. The internet of things (iot): An overview. *International Journal of Engineering Research and Applications*, 5(12):71–82, 2015. URL https://www.researchgate.net/publication/323834996_The_Internet_of_Things_IoT_An_Overview.

[13] Kebira Azbeg, Ouail Ouchetto, Said Jai Andaloussi, and Fetjah Laila. An Overview of Blockchain Consensus Algorithms: Comparison, Challenges, and Future Directions. In *Proceedings of the International Conference on Emerging Technologies and Intelligent Systems*, Advances in Intelligent Systems and Computing. Springer, 10 2020. ISBN 978-981-15-6048-4. doi: 10.1007/978-981-15-6048-4_31. URL https://www.researchgate.net/publication/344865960_An_Overview_of_Blockchain_Consensus_Algorithms_Comparison_Challenges_and_Future_Directions.

[14] David Evans, Vladimir Kolesnikov, and Mike Rosulek. Defining multi-party computation. In *A Pragmatic Introduction to Secure Multi-Party Computation*, chapter 2. NOW Publishers, 2018. URL https://securecomputation.org/docs/ch2-definingmpc.pdf.

[15] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1996. URL https://galois.azc.uam.mx/mate/propaganda/Menezes.pdf.

[16] Amos Beimel. Secret-sharing schemes: A survey. pages 11–46, 05 2011. ISBN 978-3-642-20900-0. doi: 10.1007/978-3-642-20901-7_2. URL https://www.researchgate.net/publication/220776045_Secret-Sharing_Schemes_A_Survey.

[17] Ivan Damgard, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Paper 2011/535, 2011. URL https://eprint.iacr.org/2011/535.pdf.

[18] Ivan Damgard, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority -or: Breaking the spdz limits. In *European Symposium on Research in Computer Security (ESORICS)*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013. URL https://link.springer.com/chapter/10.1007/978-3-642-40203-6_1.

[19] Ronald Cramer, Ivan Damgard, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spdz2k: Efficient MPC mod 2k for dishonest majority. Cryptology ePrint Archive, Paper 2018/482, 2018. URL https://eprint.iacr.org/2018/482.

[20] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer-Verlag, 1992. URL https://link.springer.com/content/pdf/10.1007/3-540-46766-1_34.pdf.

[21] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020. URL https://ieeexplore.ieee.org/document/9083958.

[22] National Institute of Standards and Technology. Advanced encryption standard (aes). FIPS Publication 197, NIST, May 2023. URL https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf.

[23] National Institute of Standards and Technology. Recommendation for block cipher modes of operation: Methods and techniques. Special Publication 800-38A, NIST, Dec 2001. URL https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38a.pdf.

[24] National Institute of Standards and Technology. Secure hash standard (shs). FIPS Publication 180-4, NIST, Aug 2015. URL https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf.

[25] Peter Szilagyi. Eip-225: Clique proof-of-authority consensus protocol, 2017. URL https://eips.ethereum.org/EIPS/eip-225.

[26] ConsenSys Software Inc. Besu ethereum client, 2025. URL https://besu.hyperledger.org/. Accessed June 30, 2025.

[27] Roel Wieringa. Design science methodology: Principles and practice. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '10. ACM, 2010. doi: 10.1145/1810295.1810446. URL https://ris.utwente.nl/ws/portalfiles/portal/6133973/Wieringa_2010_Design_science_methodology_principles_and_practice.pdf.

[28] Arzoo Miglani, Neeraj Kumar, Vinay Chamola, and Sherali Zeadally. Blockchain for Internet of Energy management: Review, solutions, and challenges. *Computer Communications*, 151:395–418, 2020. ISSN 0140-3664. doi: https://doi.org/10.1016/j.comcom.2020.01.014. URL https://www.sciencedirect.com/science/article/pii/S0140366419314951.

[29] Tri Nguyen, Huong Nguyen, and Tuan Nguyen Gia. Exploring the integration of edge computing and blockchain IoT: Principles, architectures, security, and applications. *Journal of Network and Computer Applications*, 226:103884, 2024. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2024.103884. URL https://www.sciencedirect.com/science/article/pii/S1084804524000614.

[30] S. Zafar, K. M. Bhatti, M. Shabbir, F. Hashmat, and A. H. Akbar. Integration of blockchain and Internet of Things: challenges and solutions. *Annals of Telecommunications*, 77:13–32, 2021. doi: 10.1007/s12243-021-00858-8. URL https://doi.org/10.1007/s12243-021-00858-8.

[31] Vinay Gugueoth, Sunitha Safavat, Sachin Shetty, and Danda Rawat. A review of IoT security and privacy using decentralized blockchain techniques. *Computer Science Review*, 50:100585, 2023. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2023.100585. URL https://www.sciencedirect.com/science/article/pii/S1574013723000527.

[32] Fateme Fathi, Mina Baghani, and Majid Bayat. Light-PerIChain: Using lightweight scalable blockchain based on node performance and improved consensus algorithm in IoT systems. *Computer Communications*, 213:246–259, 2024. ISSN 0140-3664. doi: https://doi.org/10.1016/j.comcom.2023.11.011. URL https://www.sciencedirect.com/science/article/pii/S0140366423004024.

[33] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Online: www.bitcoin.org, 2008. URL https://bitcoin.org/bitcoin.pdf.

[34] Jon Huang, Claire O'Neill, and Hiroko Tabuchi. Bitcoin Uses More Electricity Than Many Countries. How Is That Possible? The New York Times, 2021. URL https://www.nytimes.com/interactive/2021/09/03/climate/bitcoin-carbon-footprint-electricity.html. Accessed: 2024-12-08.

[35] Elizabeth Howcroft and Hannah Lang. Explainer: Ethereum's energy-saving Merge upgrade, September 2022. URL https://www.reuters.com/technology/ethereums-energy-saving-merge-upgrade-2022-09-15/. Accessed: 2024-12-08.

[36] Matthew Sparkes. Cryptocurrency Ethereum has slashed its energy use by 99.99 per cent, April 2023. URL https://www.newscientist.com/article/2369304-cryptocurrency-ethereum-has-slashed-its-energy-use-by-99-99-per-cent/. Accessed: 2024-12-08.

[37] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake, August 2012. URL https://peercoin.net/assets/paper/peercoin-paper.pdf. Whitepaper.

[38] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 02 1999. URL http://pmg.csail.mit.edu/papers/osdi99.pdf.

[39] Nikos Chondros, Konstantinos Kokordelis, and Mema Roussopoulos. On the Practicality of 'Practical' Byzantine Fault Tolerance, 10 2011. URL https://arxiv.org/abs/1110.4854.

[40] Oladayo Olufemi Olakanmi and Kehinde Oluwasesan Odeyemi. Trust-aware and incentive-based offloading scheme for secure multi-party computation in Internet of Things. *Internet of Things*, 19:100527, 2022. ISSN 2542-6605. doi: https://doi.org/10.1016/j.iot.2022.100527. URL https://www.sciencedirect.com/science/article/pii/S2542660522000294.

[41] Andrew Chi-Chih Yao. How to generate and exchange secrets. page 162–167, 1986. URL https://ieeexplore.ieee.org/document/4568207.

[42] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982. doi: 10.1109/SFCS.1982.38.

[43] Wikipedia contributors. Secure multi-party computation, 2025. URL https://en.wikipedia.org/wiki/Secure_multi-party_computation. Accessed Jan 23, 2025.

[44] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 218–229, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912217. doi: 10.1145/28395.28420. URL https://doi.org/10.1145/28395.28420.

[45] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 1–10, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912640. doi: 10.1145/62212.62213. URL https://doi.org/10.1145/62212.62213.

[46] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. Cryptology ePrint Archive, Paper 2020/521, 2020. URL https://eprint.iacr.org/2020/521.pdf.

[47] Haris Smajlović, Ariya Shajii, Bonnie Berger, Hyunghoon Cho, and Ibrahim Numanagić. Sequre: a high-performance framework for secure multiparty computation enables biomedical data sharing. *Genome Biology*, 24, 01 2023. doi: 10.1186/s13059-022-02841-5. URL https://www.researchgate.net/publication/367045174_Sequre_a_high-performance_framework_for_secure_multiparty_computation_enables_biomedical_data_sharing.

[48] Steven Kerr, Chris Robertson, Cathie Sudlow, and Aziz Sheikh. Enabling health data analyses across multiple private datasets with no information sharing using secure multiparty computation. *BMJ health and care informatics*, 32, 05 2025. doi: 10.1136/bmjhci-2024-101384. URL https://www.researchgate.net/publication/392161985_Enabling_health_data_analyses_across_multiple_private_datasets_with_no_information_sharing_using_secure_multiparty_computation.

[49] Xin Jin, Charalampos Katsis, Fan Sang, Jiahao Sun, Ashish Kundu, and Ramana Kompella. Edge security: Challenges and issues. 06 2022. doi: 10.48550/arXiv.2206.07164. URL https://www.researchgate.net/publication/361324539_Edge_Security_Challenges_and_Issues/citations.

[50] Abdul Manan Sheikh, Md. Rafiqul Islam, Mohamed Hadi Habaebi, Suriza Ahmad Zabidi, Athaur Rahman Bin Najeeb, and Adnan Kabbani. A survey on edge computing (ec) security challenges: Classification, threats, and mitigation strategies. *Future Internet*, 17(4), 2025. ISSN 1999-5903. doi: 10.3390/fi17040175. URL https://www.mdpi.com/1999-5903/17/4/175.

[51] International Organization for Standardization and International Electrotechnical Commission. Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. ISO/IEC Standard No. 7498-1:1994, 1994. URL https://www.iso.org/standard/20269.html.

[52] Tinshu Sasi, Arash Habibi Lashkari, Rongxing Lu, Pulei Xiong, and Shahrear Iqbal. A comprehensive survey on IoT attacks: Taxonomy, detection mechanisms and challenges. *Journal of Information and Intelligence*, 2(6):455–513, 2024. ISSN 2949-7159. doi: https://doi.org/10.1016/j.jiixd.2023.12.001. URL https://www.sciencedirect.com/science/article/pii/S2949715923000793.

[53] A Shaji George, T Baskar, and P Balaji Srikaanth. Cyber threats to critical infrastructure: assessing vulnerabilities across key sectors. *Partners Universal International Innovation Journal*, 2(1):51–75, 2024. URL https://www.researchgate.net/publication/378078435_Cyber_Threats_to_Critical_Infrastructure_Assessing_Vulnerabilities_Across_Key_Sectors.

[54] Manasvi Rizvi and Amer Taqa. A Review of IoT in Banking Industry. *ResearchGate*, 9:2319–5045, 09 2020. URL https://www.researchgate.net/publication/372188477_A_Review_of_IoT_in_Banking_Industry.

[55] D. Shanthi Chelliah, V. Parthiban, S. Bharathi, and R. Naveen Kumar. IoT-Based Security System for Bank Vaults. *RestPublisher*, 2(4), 2022. URL https://restpublisher.com/wp-content/uploads/2022/12/10.46632-daai-2-4-17-2.pdf.

[56] Sanjoy Mondol, Weining Tang, and Sakib Hasan. A Case Study of IoT-Based Biometric Cyber Security Systems Focused on the Banking Sector. *ResearchGate*, 03 2023. URL https://www.researchgate.net/publication/369245820_A_Case_Study_of_IoT_Based_Biometric_Cyber_Security_Systems_Focused_on_the_Banking_Sector.

[57] Marie Baezner and Patrice Robin. Stuxnet. Technical Report 4, Center for Security Studies (CSS), ETH Zurich, Zurich, October 2017. URL https://doi.org/10.3929/ethz-b-000200661.

[58] Wikipedia contributors. Python (programming language), 2025. URL https://en.wikipedia.org/wiki/Python_(programming_language). Accessed June 23, 2025.

[59] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx, and tcl for a search/string-processing program. 04 2000. URL https://www.researchgate.net/publication/36451181_An_empirical_comparison_of_C_C_Java_Perl_Python_Rexx_and_Tcl_for_a_searchstring-processing_program.

[60] TIOBE Software. The python programming language. TIOBE Index, June 2025. URL https://www.tiobe.com/tiobe-index/python/. Accessed June 23, 2025.

[61] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. IBM, jul 2014. URL https://dominoweb.draco.res.ibm.com/reports/rc25482.pdf.

[62] Ravi Patel. Understanding docker architecture: A comprehensive guide, feb 2024. URL https://medium.com/@ravipatel.it/understanding-docker-architecture-a-comprehensive-guide-5ce9129df1a4. Accessed June 23, 2025.

[63] Bitovi Academy. What is docker. URL https://www.bitovi.com/academy/learn-docker/what-is-docker.html. Accessed June 23, 2025.

[64] GitHub repository. data61/MP-SPDZ: Versatile framework for multi-party computation. URL https://github.com/data61/MP-SPDZ. Accessed: 2025-08-07.

# Appendix A

# Proof of concept prototype

## A.1 Full source code

The prototype is available at:

`https://github.com/cosmic7159/pp-iot-fault-detection-poc`

## A.2 Prototype architecture

The system consists of microservices that are connected through a local Docker network:

- **IoT simulator**: Generates synthetic fault events for cameras, motion sensors, door locks, and temperature sensors distributed across three branches.
- **Coordinator**: The coordinator is responsible for receiving faults, classifying their severity, encrypting and storing logs off-chain, forwarding branch events to the MPC system, and storing hashes or summaries on the private blockchain.
- **MPC layer**: Three MP-SPDZ parties collaboratively compute an aggregate global threat score, specifically the average severity, without the need to exchange raw logs. The service stores the result on the private blockchain.
- **Private blockchain**: A three-node Besu (Proof of Authority) network used to store log hashes and MPC results for auditability.
- **Minimal web UIs**: Dashboards designed for simulator control, coordinator score visualization, and blockchain transaction analysis.

### A.2.1 End-to-end data flow (simplified)

1. The IoT simulator produces a fault event, complete with timestamps and location metadata.
2. The coordinator receives the fault and sends the per-branch subset to the relevant MPC node.
3. MPC parties input their local aggregates to compute a global threat score through secure computation.
4. The coordinator encrypts the complete log, stores this off-chain and commits a hash to the private blockchain to serve as an integrity validator.

# Appendix B

# Data collection

This appendix describes how the data is collected in the proof of concept. All timings and resource metrics are gathered via the script collect_data.py. The script runs on the docker host and runs a predefined fault sequence from a csv file, interacting with the full system (running the predefined actions in the csv). At the end of the sequence, the script queries logs from the coordinator and blockchain services, it saves docker stats, and writes consolidated csv reports which can be used for analysis.

## B.1 Host system configuration

The system runs on a type 1 hypervisor (VMware ESXi, 8.0.3, build 24677879) that runs directly on the host hardware, avoiding the need for a general-purpose operating system in between. This setup introduces minimal additional software layers. All services run within a single Ubuntu 24.04 virtual machine (installed with the minimal configuration option) on this hypervisor. The only added packages in this virtual machine are Docker, Docker Desktop and Python3.

The proof-of-concept prototype is deployed on the following host environment. These details are captured using Python's `platform` module.

```
{
  "system": "Linux",
  "node": "ubuntu-x64-1",
  "release": "6.11.0-26-generic",
  "version": "#26~24.04.1-Ubuntu SMP PREEMPT_DYNAMIC Thu Apr 17 19:20:47 UTC 2
      ↪ ",
  "machine": "x86_64",
  "cpu_count": 16,
  "memory_total_mb": 15988.12,
  "cpu_model": "amd ryzen 7 5800x 8-core processor",
  "cpu_flags": "fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
      ↪ pat pse36 clflush mmx fxsr sse sse2 syscall nx mmxext fxsr_opt pdpe1gb
      ↪ rdtscp lm constant_tsc rep_good nopl xtopology tsc_reliable
      ↪ nonstop_tsc cpuid extd_apicid tsc_known_freq pni pclmulqdq ssse3 fma
      ↪ cx16 sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand
      ↪ hypervisor lahf_lm svm extapic cr8_legacy abm sse4a misalignsse 3
      ↪ dnowprefetch osvw topoext ssbd ibrs ibpb vmmcall fsgsbase bmi1 avx2
      ↪ smep bmi2 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt
      ↪ xsavec xgetbv1 xsaves user_shstk clzero wbnoinvd arat npt svm_lock
      ↪ nrip_save vmcb_clean flushbyasid decodeassists umip pku ospke vaes
      ↪ vpclmulqdq rdpid overflow_recov succor fsrm"
}
```

## B.2 Fault triggering logic

In the data collect_data.py collection script, the run_controlled_sequence function reads a csv of events with time offsets (the time the script waits until each new action). it waits until each offset, then posts a fault to the coordinator. if the row requests an mpc calculation it also calls the global_mpc_score endpoint. This triggers the MPC system to calculate the global threat score.

```
def run_controlled_sequence(sequence_csv):
"""
Reads a CSV of faults: each row has 'time_offset,location,device_id,fault_type,
    ↪ severity,sublocation'.
We wait until 'time_offset' seconds from the start, then trigger the fault.
"""
start_time = time.time()
rows = []

with open(sequence_csv, "r", encoding="utf-8") as f:
         reader = csv.DictReader(f)
         for row in reader:
            # Convert time_offset to float
            row["time_offset"] = float(row.get("time_offset", 0))
            rows.append(row)

# Sort by time_offset ascending
rows.sort(key=lambda x: x["time_offset"])

...

if elapsed >= offset:

    ...

    trigger_fault(loc, dev_id, ftype, sev, subloc)
    triggered = row.get("trigger_mpc_calculation", "").lower() in ("1", "true",
       ↪  "yes")
```

Listing B.1: run_controlled_sequence: scheduling and triggering faults

## B.3 Generation of sequences

We include a Python script, `generate_sequence.py`, that builds three CSV files with synthetic fault events. The script creates a `sequences` folder to store the output CSVs. These faults are used to test the system. Each file contains a synthetic fault sequence used for testing:

1. **LIGHT** (`light.csv`)
2. **MEDIUM** (`medium.csv`)
3. **HEAVY** (`heavy.csv`)

### B.3.1 Fault sampling

The function `choose_fault_with_severity()` picks a fault with the right severity:

1. Expand `SEVERITY_WEIGHTS` into a weighted list of severity levels.
2. Pick one severity at random.
3. Find all fault types in `FAULT_TYPES` with that severity. If none match, use all faults.
4. Choose one fault name from that list.
5. From the fault's category, pick a device ID in `DEVICES`.
6. Pick one allowed sublocation for that fault.

### B.3.2 Sequence generation

The function `generate_sequence(...)` builds a list of event dictionaries:

- For each of the `num_events`:
  - Sample `time_offset`:
    - With probability `burst_weight`, pick a random `burst_center` and add a uniform offset in $[-\texttt{burst\_radius}, \texttt{burst\_radius}]$.
    - Otherwise pick uniformly in $[0, \texttt{TOTAL\_SECONDS}]$.
  - Call `choose_fault_with_severity()`.
  - Pick a random branch from `BRANCHES`.
  - Decide `trigger_mpc_calculation`:
    - It must be at least `allow_mpc_every` seconds since the last true.
    - Then do a probability test: 80% inside a burst, 5% outside.
- Store each event as a dict with keys: `time_offset`, `location`, `device_id`, `fault_type`, `severity`, `sublocation`, `trigger_mpc_calculation`.
- Sort all rows by `time_offset` before returning.

## B.4 Docker stats

Along with the system logs the docker stats are also captured. We do this by connecting to the docker socket on the host and polling container metrics. This allows for the analysis of the overall system performance in each container. It does the following:

1. Connects to the Docker daemon.
2. Lists running containers with size info.
3. For each container:
   - Reads total CPU usage (nanoseconds) and converts to seconds.
   - Reads memory usage (bytes) and converts to megabytes.
   - Reads root-fs sizes (bytes) and converts to megabytes.
4. Builds a record with these fields: `timestamp`, `container_id`, `container_name`, `cpu_seconds`, `mem_mb`, `size_rw_mb`, `size_root_fs_mb`, `size_total_mb`.
5. Appends the record to an in-memory list for later export.

```
def collect_docker_stats():
"""
Gather Docker container CPU, memory, and disk-usage stats.
Stores each snapshot in DOCKER_DATA and appends to DOCKER_STATS_LOG.
"""
now_str = datetime.datetime.now(datetime.timezone.utc).isoformat()

start = time.perf_counter()

# Connect via low-level client
api = docker.APIClient(base_url=RUNNING_DOCKER_SOCK, version="auto")
try:
    api.ping()
...

# For each container, collect CPU, memory, and disk usage
for info in infos:
    cid_full = info.get("Id", "")
```

```
    short_id = cid_full[:12]
    ...

    # CPU & memory via stats()
    cpu_s = None
    mem_mb = None
    try:
        stats = api.stats(container=cid_full, stream=False)
        cpu_ns = stats["cpu_stats"]["cpu_usage"]["total_usage"]
        cpu_s = cpu_ns / 1e9
        mem_mb = stats["memory_stats"]["usage"] / (1024**2)
    except Exception as e:
        logging.warning(f"Stats error for container {name}: {e}")

    # Disk usage fields from the same containers() call
    size_rw_mb   = info.get("SizeRw",    0) / (1024**2)
    size_root_mb = info.get("SizeRootFs", 0) / (1024**2)
    size_total_mb = size_rw_mb + size_root_mb

    # Build row and store under lock
    row = {
        "data_type": "docker_stats",
        "timestamp": now_str,
        "container_id": short_id,
        "container_name": name,
        "cpu_seconds": round(cpu_s, 3) if cpu_s is not None else None,
        "mem_mb": round(mem_mb, 2) if mem_mb is not None else None,
        "size_rw_mb": round(size_rw_mb, 2),
        "size_root_fs_mb": round(size_root_mb, 2),
        "size_total_mb": round(size_total_mb, 2),
    }
    ...
```

Listing B.2: The core of the collection and processing logic for container metrics

## B.5 Final report generation

After the test sequence completes, the script stores all collected data into the following report files under `collected-data/ folder`:

1. host_info.json: JSON with OS, CPU model, core count, memory, and Linux CPU flags. The example output is shown in appendix B.1.
2. report_YYYYMMDD_HHMMSS.csv: Combined coordinator logs. The example output is shown in appendix B.6.1.
3. docker_report_YYYYMMDD_HHMMSS.csv: Docker stats snapshots. The example output is shown in appendix B.6.2.
4. blockchain_report_YYYYMMDD_HHMMSS.csv (if enabled): Blockchain transactions. The example output is shown in appendix B.6.3.
5. mpc_report_YYYYMMDD_HHMMSS.csv (if enabled): MPC-calls. The example output is shown in appendix B.6.4.

## B.6 Sample collected data outputs

This section shows example outputs generated by the data collection script. These files are available in the repository under `collected-data` and show the format of the final reports.

### B.6.1 Coordinator report example

The main coordinator report consolidates per-fault timings and metadata.

```
data_type,device_id,fault_type,location_branch,location_sublocation,ms_
    ↪ coordinator_received,ms_coordinator_forwarded,ms_sent_from_coordinator_to
    ↪ _mpc,ms_saved_by_branch_mpc_system,ms_blockchain_stored,ms_total_
    ↪ lifecycle,timestamp_collected
coordinator_log,base_device,base_fault,Branch1,base_subloc,0,0,0,9,49,49,2025-0
    ↪ 6-14T11:38:51.637685+00:00
coordinator_log,base_device,base_fault,Branch2,base_subloc,0,0,0,5,20,20,2025-0
    ↪ 6-14T11:38:51.674487+00:00
coordinator_log,base_device,base_fault,Branch3,base_subloc,0,0,0,5,18,18,2025-0
    ↪ 6-14T11:38:51.698440+00:00
coordinator_log,camera2,camera_blurred,Branch1,Parking-lot,0,0,0,3,15,15,2025-0
    ↪ 6-14T11:39:06.737778+00:00
...
```

Listing B.3: collected-data/examples/report_20250614_133851.csv

### B.6.2 Docker stats example

All docker stats are recorded here.

```
data_type,timestamp,container_id,container_name,cpu_seconds,mem_mb,size_rw_mb,
    ↪ size_root_fs_mb,size_total_mb
docker_stats,2025-06-14T11:38:51.576597+00:00,7a8373a54d9e,iot-simulator,0.406,
    ↪ 34.5,0.24,1059.53,1059.77
docker_stats,2025-06-14T11:38:51.576597+00:00,34cecc2e674c,mpc_node_1,0.425,31.
    ↪ 5,0.01,1073.69,1073.7
docker_stats,2025-06-14T11:38:51.576597+00:00,1d371c97ff2a,mpc_node_2,0.456,32.
    ↪ 1,0.01,1073.69,1073.7
docker_stats,2025-06-14T11:38:51.576597+00:00,1c7bdfe5ab86,mpc_node_0,0.428,33.
    ↪ 88,0.01,1073.69,1073.7
...
```

Listing B.4: collected-data/examples/docker_report_20250614_133851.csv

### B.6.3 Blockchain report example

All on-chain transactions (hashes and scores) are recorded here.

```
data_type,block_timestamp,block_number,transaction_data
blockchain_tx,2025-06-14T11:38:52+00:00,13,b0219e25f14b89eed9ccbaaac96e51dfd003
    ↪ ceb9fb9d5bcb67734861e7c593ab
blockchain_tx,2025-06-14T11:38:52+00:00,13,28471c82710733232534c22db95790b993fd
    ↪ 9bb1c215705660253f017dcf3fb6
blockchain_tx,2025-06-14T11:39:07+00:00,14,d90f0d6c66c51e6f14e9dc6a486c834194f3
    ↪ 4c1abe1a6533ff8e5af9c5e417a6
blockchain_tx,2025-06-14T11:39:07+00:00,14,347f941121c9274c267a62159c5f1d2b93d7
    ↪ f7dbf63d2a6428d815d889bc7a17
blockchain_tx,2025-06-14T12:38:52+00:00,253,Global Threat Average = 31.45%
...
```

Listing B.5: collected-data/examples/blockchain_report_20250614_133851.csv

### B.6.4 MPC call records example

Each MPC invocation's timing and results are captured here.

```
branch,device_id,triggered,start_timestamp,end_timestamp,ms_mpc_call,status,
    ↪ data
Branch1,camera2,False,2025-06-14T11:39:06.743051+00:00,2025-06-14T11:39:06.7430
    ↪ 51+00:00,,,
```

```
Branch3,motion2,False,2025-06-14T11:39:24.779604+00:00,2025-06-14T11:39:24.779604+00:00,,,
Branch2,motion2,False,2025-06-14T11:39:25.803450+00:00,2025-06-14T11:39:25.803450+00:00,,,
Branch1,temperature1,True,2025-06-14T12:38:40.756853+00:00,2025-06-14T12:38:45.344364+00:00,4587,200,"{'blockchain_response': {'transactionHash': '6b8fe62ae51770049ca4f0e6d2d19a2d1428ca71f25fc6055fc677109b91fd75'}, 'global_threat_score': 31.44714}"
...
```

Listing B.6: collected-data/examples/mpc_report_20250614_133851.csv

# Appendix C

# MPC implementation

This appendix provides a summary of the MPC component in the prototype system. The MPC implementation consists of a python wrapper, which orchestrates the MPC executables (MP-SPDZ), and a compilation script which builds these executables the wrapper interacts with. The MPC calculations are performed with MP-SPDZ. We use the SPDZ$^{2k}$ protocol.

## C.1 Overview

The MPC component of the PoC implements a three-party secret sharing system using MP-SPDZ to compute a weighted average of fault severities. Each party holds the following data, which is used to calculate the overall average severity:

- A sum of accumulated severity at a local level.
- A local event count.

## C.2 Build and deployment

### C.2.1 Building MP-SPDZ artifacts

Before the MPC system can be used, the binaries that handle the MPC protocols have to be compiled. If the binaries do not exist on the host machine, the helper script `mpc/build_mpc_binaries.sh` starts a Docker container that:

1. Clones the MP-SPDZ source code.
2. Builds `spdz2k-party.x` (online binary) and `Fake-Offline.x`.
3. Compiles the `threat_score.mpc` program.
4. Performs offline preprocessing.

## C.3 Service API

- The service API is in run_mpc.py (them python wrapper).
- The received branch-level faults are stored on the respective local MPC system.
- `GET /compute_mpc`:
    1. Reads local logs -> computes (sum, count).
    2. Scales sum by $10^6$ for fixed-point.
    3. Runs `spdz2k-party.x` interactively:

```
cmd_stage2 = [
    MP_SPDZ_BIN,
    "-v", # verbose
    "-I", # Using -I activates interactive mode (necessary in order to
        ↪ reveal the output to ALL parties)
    "-R", str(BITLEN), # ring mode
    "-N", str(TOTAL_NODES), # number of parties
    "-S", str(SECPAR), # must match the Fake-Offline.x invocation
    "-h", first_host, # only the host for party 0
    "-pn", "14000", # port number
    str(NODE_INDEX), # this party's index
    MPC_PROGRAM # .mpc bytecode name
]
...

try:
    proc = subprocess.run(
        cmd_stage2,
        ...
```

4. Parses returned secret result and converts to percentage.

- `GET /global_threat` handles `/compute_mpc` calls to all three parties, verifies consistency, and posts the agreed score to the blockchain.

## C.4 MPC program: `threat_score.mpc`

The MPC program that is used in MP-SPDZ:

```
from Compiler.types import sint
from Compiler.library import print_ln

# Accumulate total severity and total count
total_sev = sint(0)
total_cnt = sint(0)

# Each party provides two inputs in sequence:
# 1) sum of its local severity scores
# 2) number of events it contributed
for p in range(3):
    # Read the aggregated severity sum and count from party p
    sev_sum = sint.get_input_from(p)
    cnt = sint.get_input_from(p)

    total_sev += sev_sum
    total_cnt += cnt

# Compute the integer average: total_sev / total_cnt
# Use a safe bit-length
avg_sev = total_sev.int_div(total_cnt, 24)

# Reveal and print just the numeric average
print_ln('%s', avg_sev.reveal())
```