COMP997

Dissertation

Master of Cyber Security & Digital Forensics

Utilising the Zero Knowledge Proofs (ZKPs) technique over Hyperledger Fabric, a permissioned open-source blockchain architecture, to create an anonymous cross smart contract authentication mechanism prototype, enabling smart contract interoperability

Supervisor: Shu Su

Co-supervisor: Jeff Nijsse

Benjamin Oholeguy

Semester 2 - 2024

Table of Contents

[Table of Figures 4](#_Toc179881854)

[Abstract 6](#_Toc179881855)

[1. Introduction 6](#_Toc179881856)

[1.1. Motivation 8](#_Toc179881857)

[1.2. Contributions 8](#_Toc179881858)

[2. Literature review 8](#_Toc179881859)

[3. Research Design 10](#_Toc179881860)

[3.1. Research Questions 10](#_Toc179881861)

[3.2. Defining the prototype 10](#_Toc179881862)

[4. Prototype Setup 13](#_Toc179881863)

[4.1. Zero Knowledge Proof 13](#_Toc179881864)

[4.1.1. Implementation 13](#_Toc179881865)

[4.1.2. Circuit definition 15](#_Toc179881866)

[4.2. SnarkJS installation set-up process 16](#_Toc179881867)

[4.3. Smart-contract Application 16](#_Toc179881868)

[4.3.1. Apartment Smart Contract 16](#_Toc179881869)

[4.3.2. Energy Smart Contract 17](#_Toc179881870)

[4.3.3. Common for both Smart Contract 17](#_Toc179881871)

[4.4. TypeScript Applications 17](#_Toc179881872)

[4.4.1. Apartment TypeScript App 17](#_Toc179881873)

[4.4.2. Energy TypeScript App 18](#_Toc179881874)

[4.5. The Oracle 18](#_Toc179881875)

[5. Prototype Execution 18](#_Toc179881876)

[5.1. Explanation 19](#_Toc179881877)

[5.2. Plonk 19](#_Toc179881878)

[5.3. Groth16 19](#_Toc179881879)

[5.4. Fflonk 20](#_Toc179881880)

[5.5. The Oracle 20](#_Toc179881881)

[6. Infrastructure 20](#_Toc179881882)

[6.1. Hyperledger Fabric installation 20](#_Toc179881883)

[6.2. Smart-contract installation (deploy) 21](#_Toc179881884)

[7. Results and Discussion 22](#_Toc179881885)

[7.1. Research Q2 answer 22](#_Toc179881886)

[7.2. Data Analysis 23](#_Toc179881887)

[7.2.1. Plonk 23](#_Toc179881888)

[7.2.2. Groth16 25](#_Toc179881889)

[7.2.3. Fflonk 26](#_Toc179881890)

[7.3. Research Q3 answer 27](#_Toc179881891)

[8. Conclusion 28](#_Toc179881892)

[8.1. Limitations 28](#_Toc179881893)

[8.2. Further research 28](#_Toc179881894)

[9. References 30](#_Toc179881895)

[10. Acknowledgements 32](#_Toc179881896)

[11. Appendix 32](#_Toc179881897)

[11.1. The circuit 32](#_Toc179881898)

[11.2. SnarkJS 33](#_Toc179881899)

[11.3. Smart contracts 34](#_Toc179881900)

[11.4. Node applications 36](#_Toc179881901)

[11.5. ZKPs Prototype execution 38](#_Toc179881902)

[11.6. The oracle. Prototype execution 42](#_Toc179881903)

# Table of Figures

[Figure 1. Interaction diagram between energy and apartment smart contracts. Apartment smart contract uses SnarkJS library to do the verification. 12](#_Toc179998842)

[Figure 2. Sequence diagram of the interaction between energy and apartment smart contract. The apartment smart contract uses SnarkJS library to do the verification process. After that, if the answer is yes, it would send the data back to the energy smart contract. 13](#_Toc179998843)

[Figure 3. Sequence diagram interaction between the energy smart contract, the energy node application and the Accuweather API. 13](#_Toc179998844)

[Figure 5. Hyperledger Fabric entities to support the dissertation prototype. 14](#_Toc179998845)

[Figure 4. Circuit definition using circom for the prototype. 17](#_Toc179998846)

[Figure 6. Histogram of the three protocols by milliseconds. 23](#_Toc179998847)

[Figure 7. Plonk protocol performance data after running 10,000 queries. 24](#_Toc179998848)

[Figure 8. Plonk protocol performance data after running 10,000 queries grouped by milliseconds. 24](#_Toc179998849)

[Figure 9. Groth16 protocol performance data after running 10,000 queries. 25](#_Toc179998850)

[Figure 10. Groth16 protocol performance data after running 10,000 queries grouped by milliseconds. 26](#_Toc179998851)

[Figure 11. Fflonk protocol performance data after running 10,000 queries. 26](#_Toc179998852)

[Figure 12. Fflonk protocol performance data after running 10,000 queries grouped by milliseconds. 27](#_Toc179998853)

[Figure 13. The expected cube is exposed to release the pubic signals. 33](#_Toc179998854)

[Figure 14. Verification key output. This value was hard coded in the smart contract for ZKPs verification. 33](#_Toc179998855)

[Figure 15. File public.json with the public signal values. 34](#_Toc179998856)

[Figure 16. File proof.json generated storing the proof value to be used at the smart contracts for ZKPs verification. 34](#_Toc179998857)

[Figure 17. File input.json with the public and private inputs. 34](#_Toc179998858)

[Figure 18. Require JavaScript functionality to include a module in the code. 35](file:////Users/benjaminoholeguy/Documents/personal/master_of_cyber_security_and_digital_forensics/thesis/documentation/Oholeguy.docx#_Toc179998859)

[Figure 19. The energy and apartment smart-contracts configuration file package.json is shown with snarkjs library added as a dependency. 35](#_Toc179998860)

[Figure 20. Energy smart contract CreateAsset defines the apartment structure. 35](#_Toc179998861)

[Figure 21. Apartment smart contract function QueryEnergyUsage returns the energy used in each apartment. 35](#_Toc179998862)

[Figure 22. CreateAsset function in Energy smart contract. 36](#_Toc179998863)

[Figure 23. GetEnergyUsage in energy smart contract. 36](#_Toc179998864)

[Figure 24. VerifyPlonk function code. 36](#_Toc179998865)

[Figure 25. VerifyGroth16 function code. 36](#_Toc179998866)

[Figure 26. VerifyFflonk function code. 36](#_Toc179998867)

[Figure 27. NodeJS application createAsset function for creating apartments with energy usage random integer value in apartment smart contract. 37](#_Toc179998868)

[Figure 28. NodeJS application createAsset function for creating apartments with energy usage value equal 0 in energy smart contract. 37](#_Toc179998869)

[Figure 29. Energy NodeJS application updateAsset triggers the process of fetching energy usage from apartment smart contract. 38](#_Toc179998870)

[Figure 30. Energy NodeJS application oracleUpdateAsset function. 38](#_Toc179998871)

[Figure 31. Energy NodeJS application CallOracle function. 38](#_Toc179998872)

[Figure 32. Energy smart contract OracleUpdateAsset function. 39](#_Toc179998873)

[Figure 33. Apartment smart contract CreateAsset function terminal output. 39](#_Toc179998874)

[Figure 34. Energy smart contract function CreateAsset with EnergyUsage attribute equals 0. 40](#_Toc179998875)

[Figure 35. Energy smart contract UpdateAsset function output. 40](#_Toc179998876)

[Figure 36. Energy smart contract UpdateAsset terminal output execution showing the asset with the lowest latency verifying with the Plonk protocol. 41](#_Toc179998877)

[Figure 37. Energy smart contract UpdateAsset terminal output execution showing the asset with the highest latency verifying with the Plonk protocol. 41](#_Toc179998878)

[Figure 38. Energy smart contract UpdateAsset terminal output summary using the Plonk protocol. 41](#_Toc179998879)

[Figure 39. Energy smart contract UpdateAsset terminal output execution showing the asset with the lowest latency verifying with the Groth16 protocol. 41](#_Toc179998880)

[Figure 40. Energy smart contract UpdateAsset terminal output execution showing the asset with the highest latency verifying with the Groth16 protocol. 41](#_Toc179998881)

[Figure 41. Energy smart contract UpdateAsset terminal output summary using the Groth16 protocol. 42](#_Toc179998882)

[Figure 42. Energy smart contract UpdateAsset terminal output execution showing the asset with the lowest latency verifying with the Fflonk protocol. 42](#_Toc179998883)

[Figure 43. Energy smart contract UpdateAsset terminal output execution showing the asset with the highest latency verifying with the Fflonk protocol. 42](#_Toc179998884)

[Figure 44. Energy smart contract UpdateAsset terminal output summary using the Fflonk protocol. 42](#_Toc179998885)

[Figure 45. OracleUpdateAsset terminal output execution with Accuweather API attribute TypeID equals 0. 43](#_Toc179998886)

[Figure 46. OracleUpdateAsset terminal output execution with Accuweather API attribute TypeID equals 1. 43](#_Toc179998887)

# Abstract

As an emergent concept, a smart city represents a way to organise and improve the use of resources. Optimising time and energy or providing health services with better performance can comfort its inhabitants. Additionally, a smart city’s resource management positively impacts the environment. Internet of Things, wearable devices, and sensors create an ecosystem where data is constantly generated, shared, and analysed to make decisions. However, these benefits also represent the challenge of information management. Anonymity, confidentiality, integrity, and information availability are paramount for smart cities. By testing three different proof verification protocols, Plonk, Groth16 and Fflonk, this research aims at implementing a ZKPs-based piece of software (gateway) to interconnect more than one chaincode (smart contract) for sharing information, running on the same Hyperledger Fabric (Fabric) blockchain. This study will also focus on unveiling the advantages and disadvantages of sharing information in a smart city environment.

After running three tests, one for each protocol, the results have shown that the novel Fflonk protocol performs better, with a latency average time of 20.535 milliseconds to verify each query. Groth16 with 22.523 milliseconds and Plonk with 24.35 milliseconds followed in that order.

The Fabric blockchain smart contracts, the Node.js gateway application and the three ZKPs protocols set-up, Groth16, Plonk and Fflonk, used in this dissertation, are version-controlled and available on GitHub[[1]](#footnote-1).

# Introduction

Security is paramount in a smart city environment as the population grows and urbanisation becomes more complex. Records generated by and for their inhabitants and all associated devices push the limits of data recollection and processing performance and security.

As an advanced vision of the future, smart cities are rising in this digital era as a vision of urban development, where advanced technology and city infrastructure become integrated with the goal of improving their inhabitants’ life quality (Zhang et al., 2017). Smart transport, sensor networks for monitoring, energy resources management, and digitalised public services are just some of these technologies (Eskhita et al., 2021). However, identification, recollection and analysis (Javed et al., 2022) of all the inhabitants’ generated data raises concerns about user privacy and the security of the information (Farayola et al., 2024) because of the threat of potential attacks (Ahmid & Kazar, 2023; Arogundade, 2023).

Since users have to share information to make the smart city ecosystem work, they have to trust the process, and that is the reason why user privacy is mandatory. Data they generate contains sensitive personal information such as behaviour and consumption patterns, or health information. Moreover, the user’s data visibility and misuse of it by attackers would result in a loss of trust in the smart city system by the user. Consequently, it is essential to implement security and privacy data protection measures to guarantee the secure and ethical use of this information (Rao & Deebak, 2023).

The necessity of a reliable storage mechanism for the massive amount of data generated links with the blockchain network, a technology conceived fundamentally as a public utility aiming to create a virtual analogue to physical currency. Nakamoto initially delineated the advent of blockchain technology in a pivotal manuscript entitled “Bitcoin: A Peer-to-Peer Electronic Cash System” (Nakamoto, 2008), which introduced a transformative paradigm for the authentication of transactions. Contrasting with extant methodologies that depended on centralised validation mechanisms, this approach removes the necessity for a central validating authority (Enisa, 2023).

The architectural design of the blockchains underscored the profound potential inherent in decentralisation. The underlying tenets set forth by the Bitcoin protocol have exerted a considerable influence on the subsequent development of many alternative cryptocurrencies. One of those cryptocurrencies is Ethereum, which has extended the capabilities of the blockchain infrastructure by using smart contracts (Buterin, 2014), also called chaincode in the Fabric blockchain context (Fabric, 2023), the infrastructure used in the prototype for this dissertation. These entities transcend mere transactional functions, serving as programmable scripts capable of autonomously enforcing predefined logical parameters and executing operations. As the parameters of decentralisation and automation have broadened, there has been a concomitant expansion in the potential applicative domains for blockchain-enabled solutions, culminating in a substantial proliferation within this field (Zarrin et al., 2021).

Despite its numerous advantages, blockchain technology is not without its limitations. Many of these are inherent to the architecture and implemented to ensure alignment with the system’s fundamental requirements. Perhaps the biggest limitation is the blockchain’s inherent design as a distributed ledger system. This necessitates that each node within the network retain a complete copy of all the information held by the other nodes, a redundancy that is indispensable for maintaining the system’s decentralised characteristics. Every node must possess the capacity to autonomously perpetuate the network’s functionality, a feature that entails a trade-off, manifesting as restrictions on the amount of data that each transaction can encompass, as well as limitations on the chain’s capacity to store only the most essential data required for transaction processing. This intrinsic data redundancy consequently imposes constraints on the functional capabilities of smart contracts, which are predominantly confined to interacting with the data available within the blockchain and the transaction, further compounded by the associated costs incurred by the storage of larger data quantities (Khan et al., 2020).

One consideration to be made when adopting decentralised ledger technology is security and privacy. Even though Zero Knowledge Proofs (ZKPs), the chosen protocol for the prototype implemented in this dissertation, is presented by several research papers as cutting-edge technology (Groth, 2016), there are many applications regarding user privacy-preserving techniques such as privacy-preserving identity management (Bernabe et al., 2019), privacy-preserving cross-domain authentication (Huang & Shen, 2024), privacy-preserving data analytics (Zhang et al., 2023), secure and lightweight authentication protocol (Oh et al., 2021), and Fully Homomorphic Encryption (FHE), presented by Zhang et al. (2017) as a more efficient option to the detriment of standard encryption when data has to be processed by the device. Supporting their statement, the authors express that security relies on the fact that data remains encrypted from the beginning to the end of the process and privacy for the reason that FHE maintains sensitive information hidden, even whilst being processed. The drawback is that the mathematics operations that can be done are simple.

Despite the fact that several protocols are being used and have to be considered as an option when looking for privacy, anonymity or security, it is the capacity to prove encrypted knowledge without exposing sensitive data, one of the ZKPs characteristics that made this protocol a desired feature for all applications (Ernstberger et al., 2024). Non-interactivity and succinctness are also ZKPs functionalities highlighted by the authors in their research “Do you need a Zero Knowledge Proof?”.

In addition to non-interactivity, Hasan (2019) affirms that soundness and completeness are also aspects that a zero-knowledge proof has to satisfy. Soundness means that there is a low chance but not 0, that if the secret is false, the verifier cannot evaded by the prover. Zero-knowledge relies on the idea that the verifier does not know anything other than the secret if it is true. Finally, completeness, if the secret is true, the prover will know it.

Through the next sections of the dissertation, evidence of a Hyperledger Fabric (Fabric) implementation using Circom for creating the circuit and the protocols Plonk, Groth16 and Fflonk to do the verification will be shown. A prototype to test the latency of the communication between two smart contracts will be run with 10,000 assets for each protocol.

## Motivation

Recognising the need for blockchain interconnections and understanding that not much study has been conducted, the gaps in evidential support motivated the study to contribute through this applied research project. The decision to focus on Fabric and ZKPs was made due to the extended worldwide use of this blockchain use and the ZKPs mechanism due to the information derived (Chen et al., 2023; Martinez et al., 2024; Zhou et al., 2024).

Due to the large amount of data generated in a smart city environment, the necessity of maintaining it secure and private but also doing it as fast as possible motivates this dissertation to test the latency in three ZKPs protocols, Plonk, Groth16, and Fflonk.

## Contributions

This dissertation has mainly contributed in two aspects. Firstly, determining which of the three protocols mentioned above has the fastest latency in the communication between two smart contracts running on the same blockchain and the same channel, thereby determining that Fflonk protocol is more reliable than the others.

The second contribution is the very same application. All that is developed in it, Flonk, Groth16, Plonk, the Fabric smart contracts, and the oracle, can be used, cloned, downloaded, transformed or upgraded by anyone interested in it as desired.

The third contribution is that the benefits of the proposed approach would be the reduction of energetic consumption and associated costs in the smart city, as well as efficiency, sustainability, carbon footprint, transparency, privacy and security. Building residents would enjoy better living conditions as they can get the energy price adjusted depending on the generation cost. Getting data from outside the Fabric to impact the final invoice would benefit the smart city.

# Literature review

The foundational paper on Zero Knowledge Proof of Identity (Fiege et al., 1987) defines it as a cryptographic technique to demonstrate that a message is true without revealing anything more than the indispensable information to prove it and, consequently, keeping the rest of the data, such as personal information, unrevealed, ZKP is presented as a valid decentralised solution to solve identity theft and lack of identity.

Considering the Internet of Things (IOT) as one of the most prolific data generators in a smart city context, a recent study by Chen et al. (2023) established that in practice, ZKPs communication overload produced in the IOT context makes it useless due to the massive amount of nodes. The approach proposed by the authors is the non-interactive zero-knowledge proof (NIZK). However, a study by Martinez et al. (2024) found that ZKPs can improve smart contract interoperability.

Ernstberger et al. (2024) shows applications that do not use the “zero knowledge” functionality of the protocol to prove the circuit but apply its succinctness and non-interactivity. The authors stated that a non-interactive proof avoids the posterior communication between the sender and the receiver after sending the signals, establishing that interaction proofs are not exempt from a very small probability of validating an invalid proof. The research states that any interactive proof can be non-interactively validated just using a few bits of the proof randomly selected based on the probabilistically checkable proof (PCP) theorem.

Non-interactive ZKPs are usually built for one purpose, for example, to know a secret, but they can be used to prove any type of statement. However, traditional cryptography is interactive, requiring messages to be exchanged between the sender and the receiver. Turning an interactive algorithm into a ZKPs non-interactive algorithm is a procedure called the Fiat-Shamir technique, proposed by Dagdelen et al. (2013), which turns identification into signatures schemes.

Hasan (2019) offers an explanation about non-interactive ZKPs verification using two actors, Alice as the sender and Bob as the receiver. On the one end of the communication, first, Alice knows her secret *“s”* and *PKk = gs (mod p)*. Second, a random number *“rn”* is selected, creating the following commitment *PKrn = grn*. Third, Alice calculates her own challenge *c = Hash( g || s || PKk || PKrn )*.

In contraposition with the interactive way to let Bob know that she knows the secret information *“s”*, Alice is creating her own challenge but not both. In the prototype developed for the dissertation, the Hash function for creating the challenge is executed when the creation of the witness in the file witness\_calculator.js. Finally, Alice calculates *r = v – c \* s* and sends *PKv, c, r* to Bob. When Alice sends Bob all the numbers, Bob has to save *“c”* to avoid intruders impersonating Alice.

On the other end of the communication, Bob calculates *Vverify , Vverify = gr \* (gs)c*. If Bob corroborates that *Vverify* *= PKv*, then it is okay. If the values are equal after checking the verifying equations, it means that Alice knows the secret key *“s”*.

Liu et al. (2024) express that proof generation for ZKPs is still a performance problem in this context when a large amount of data has to be processed, such as zkRollups, which can multiply by 100 times the amount of system processing and zkEVM. None of them were tested in this research. The authors presented data showing that the Plonk protocol can only scale a circuit to 225 inputs in a 200GB memory computer. Finally, several companies implementing powerful and expensive terabyte clusters are given in order to exemplify how costly proof generation is.

Buterin (2021) explains that zkRollups are a way to package several transactions, check them off-chain with ZKPs, and after that, deploy them in the blockchain. Additionally, zkEVMs is presented as a gear that allows Ethereum and ZKPs to work together privately.

Ambrona et al. (2022) in their research “*New optimization techniques for Plonk’s arithmetization*” affirms that due to its performance, Plonk is used in several environments such as Filecoin, an identification to storage, and Zcash for private transactions. In addition, the authors indicate that due to its solid proof and high-performing verification process, the protocol is appropriate for online verification in the blockchain context.

Even though Ballesteros-Rodríguez et al. (2024) in their research “Enhancing Privacy and Integrity in Computing Services Provisioning Using Blockchain and zk-SNARKs”, established that the public signals and proof created using Plonk are slower than using other zk-SNARK scheme constructions like Groth16, the goal of this dissertation is to test the performance of the three protocols, Groth16, Plonk and Fflonk.

Creating the proof requires doing the hard work off-chain and sending it to someone who can verify it quickly. The first need is a problem able to be transformed into an arithmetic circuit. After that, it is needed the extraction of the polynomial from the circuit and a commitment of the polynomial. Finally, the polynomial must be sent to the verifier. Turning a problem into a circuit can be seen in this Circom circuit shown in Figure 4.

Ambrona et al. (2022) states that permutations over Lagrange bases for Oecumenical Noninteractive arguments of Knowledge (Plonk) is a cryptographic protocol to create ZKPs emphasising on non-interactive universal solutions for zero-knowledge proofs. These proofs are efficient in verification time, representing an advantage for decentralised systems such as smart city environments. In addition, security is also paramount. For that matter, the concept of computational soundness in interactive proofs ensures that the system remains resilient against attempts by a computationally powerful and malicious prover to deceive the verifier (Ernstberger et al., 2024).

Groth (2016) proclaims the advance in cryptography to reach better but smaller and simpler to verify non-interactive arguments, named succinct non-interactive arguments of knowledge (SNARKs), as an improvement of the pairing-based succinct non-interactive arguments (SNARGs). The improvement proposed is based on the efficiency of SNARKs, given by the use of asymmetric pairing, a language NP-complete (the output to any problem would always be true or false), a proof using three group elements. Finally, SNARKs check one math equation with three pairings to do the validation.

In the research “PlonK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge” (Gabizon et al., 2019), the authors presented a way to simplify the arithmetization step and the permutation argument in Plonk, a universal SNARK non-interactive and succinct protocol in opposition with Groth16, a non-universal option.

Singh (2024) stated that the tool has three components. First, the circuit compiler allows the definition of rules and conditions to be verified without revealing the subjacent data. They can be one or more, and they are also utilised for proof generation. The second is the proof generation. After the circuit definition, the tool can generate a zk-SNARK proof to verify that the input data complies with the circuit condition. Finally, proof verification provides the necessary tools to verify the proof. The verification confirms that the hidden input data follow the conditions specified in the circuit.

Used for the implementation of the prototype in this dissertation, SnarkJS is a JavaScript-developed tool designed to work with ZKPs non-interactive. These proofs allow to demonstrate that an affirmation is true without revealing any additional information than the affirmation itself. This is especially useful in applications that require security and privacy, like blockchain transactions, password-less secure authentication or IoT environments where privacy and verification are paramount. Additionally, SnarkJS acts as a bridge between cryptographic theory and its practical application (Iden3, 2024a).

# Research Design

The literature review showed that while technologies like ZKPs, zkRollups, and zkEVMs have improved security, speed and privacy, there is still not much research on how they work in production for smart cities. It also pointed out issues with how different ZKP protocols, Groth16, Plonk, and Fflonk, perform. These gaps led to the research questions, which focus on improving the use of these technologies in smart cities.

## Research Questions

* + 1. How can a distributed and agnostic mechanism be created to establish connections between different blockchain types to interconnect different smart contracts in a smart city?
    2. How can ZKPs achieve cross-smart-contract, anonymous and low-latency transactions in a smart city context, and how are these attributes related?
    3. What are the security considerations in a smart city when implementing cross smart contract communication?

## Defining the prototype

The goal is to design and implement a prototype using the blockchain Fabric to develop and deploy two smart contracts. The smart contracts need a channel to run over. The name of this channel would be “mychannel”. The smart contract names would be Energy and Apartment. The first smart contract will represent the power company, and the second smart contract the apartment power meter readings.

The interaction between both is given by the necessity of the power company to fetch and store the power usage from each apartment power meter. Furthermore, after getting the power meter reading, the energy smart contract has the responsibility to get a blockchain-external value to define the price of the energy usage per unit.

A diagram of a computer

Description automatically generated

Figure 1. Interaction diagram between energy and apartment smart contracts. Apartment smart contract uses SnarkJS library to do the verification.

The diagram above, shown in Figure 1, presents the expected interaction between apartment and energy smart contracts to be implemented in the prototype. Additionally, it presents a ZKPs “verify” function as an entity that receives the signal from the apartment smart contract to determine whether the request is authorised.

The sequence diagram for the ZKPs verification in the apartment smart contract is presented in Figure 2. Moreover, the apartment smart contract is expected to use the library SnarkJS and the protocols Groth16, Plonk and Fflonk to do the verification in three different tests. The process would start with the energy smart contract recording the start time when requesting the energy usage from the apartment. After receiving the response, the energy smart contract would have the responsibility of recording the end time, calculating the difference between them and recording it.

A diagram of a project

Description automatically generated

Figure 2. Sequence diagram of the interaction between energy and apartment smart contract. The apartment smart contract uses SnarkJS library to do the verification process. After that, if the answer is yes, it would send the data back to the energy smart contract.

Finally, the Oracle sequence diagram shown in Figure 3 clarifies the defined process of fetching data from outside the blockchain to decide the energy price. The event is defined in the energy smart contract to trigger the off-chain function oracleUpdateAsset to fetch the data from the Accuweather API. Once the data is fetched, it would be sent back to the energy smart contract to define the price and record it in the asset.

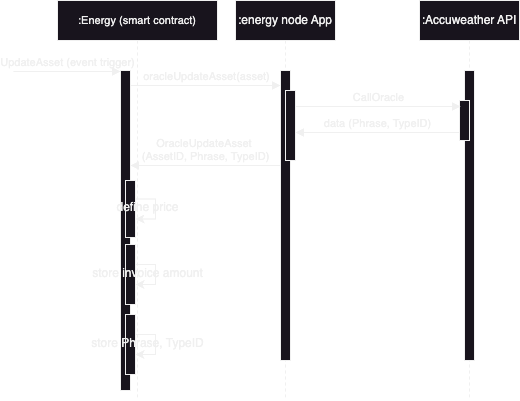


Figure 3. Sequence diagram interaction between the energy smart contract, the energy node application and the Accuweather API.

## Infrastructure

### Hyperledger Fabric installation

The prototype implemented in the dissertation began defining one local Hyperledger Fabric blockchain network running over the MacOS Sonoma operating system with Apple M1 Max Chip.

The Fabric installation consisted of, first, downloading the Fabric files using the following command:

$ curl -sSL https://raw.githubusercontent.com/hyperledger/fabric/main/scripts/bootstrap.sh | bash -s

After the files are downloaded and copied to the folder fabric-samples, move to fabric-samples/test-network to execute the following command to run the Fabric and create the channel.

$ ./network.sh up createChannel -c mychannel -ca

The Hyperledger Fabric has a test network to be executed, and nodes are run on a local machine. In this case, the script network.sh executes the test-network and creates the channel mychannel. These nodes are one ordering and two peers’ organizations and all certificates are issued by the root CA’s as shown in Figure 5.

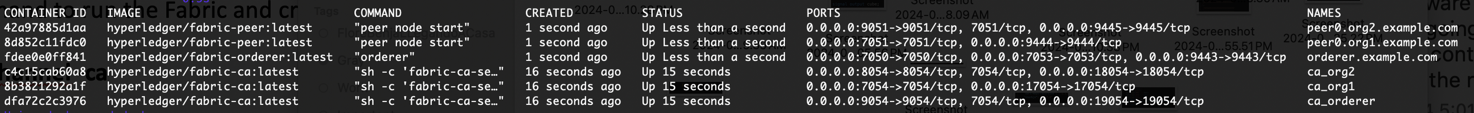


Figure 5. Hyperledger Fabric entities to support the dissertation prototype.

### Smart-contract installation (deploy)

A smart contract is a piece of code accepted by the members of the network to manage the logic. Although the name smart contract is the term widely accepted in the blockchain context, Hyperledger Fabric names them as chaincode. It is the same thing, nevertheless (Fabric, 2023).

Both smart contracts, energy and apartment, were deployed to the channel mychannel using deployCC subcommand from the test-network directory as follows:

$ ./network.sh deployCC -ccn energy -ccp ../asset-transfer-events/\_chaincode-javascript-energy -ccl javascript -ccep "OR('Org1MSP.peer','Org2MSP.peer')"

$ ./network.sh deployCC -ccn apartment -ccp ../asset-transfer-events/\_chaincode-javascript-apartment -ccl javascript -ccep "OR('Org1MSP.peer','Org2MSP.peer')"

The smart contract started after the execution of the script network.sh with the subcommand deployCC, the flag ccn to name the smart contract, the ccp flag to indicate its path, and the ccl flag the language used to code it. Finally, ccep flag deployed with an endorsement policy, allowing both organisations, Org1MSP and Org2MSP, to create assets without getting an endorsement from the other (Fabric, 2022).

# Prototype Setup

The goal of this prototype is to simulate the energy reading, simulating a smart city, from the energy grid to the apartment building management. It starts by fetching the energy usage from the apartment smart contract to be used in the energy smart contract. After fetching the energy usage from the apartment, the energy smart contract fetches forecast weather data from outside the blockchain to define the electricity price per unit of energy used.

The prototype’s actors are A) the Energy Grid Network System, represented by the energy smart contract, to manage energy transactions inside the energy network, obtaining real-time consumption. B) The Building Management System, represented by the apartment smart contract energy, generates and stores the energy usage, which would be read by the energy smart contract. C) Apartment NodeJS application to trigger the generation of the apartments and its power meter value to be read, and D) Energy NodeJS application to update the apartments in the energy smart contract reading the energy usage in the apartment smart contract. E) This NodeJS application also sets an event which triggers a request for an external value to Accuweather API, called the oracle, anytime the update asset is called.

The apartment attributes in the apartment smart contract are ApartmentID and EnergyUsage.

The apartment attributes in the energy smart contract are ApartmentID, EnergyUsage, StartTime, EndTime, Latency, Price, Phrase, TypeID and Invoice.

The apartment ledger defines N apartments to be initialised with ApartmentID and EnergyUsage as attributes. First, EnergyUsage is defined with a value equal to 0. Second, by using an external node application to trigger the prototype, this attribute is initialised with a random number representing the energy usage or the power meter of the apartment at a given moment.

The node application bellow arbitrarily creates 10,000 apartments with the attribute ApartmentID equal to the string Apartment plus the apartment’s number e.g.: “Apartment3” and EnergyUsage equal to a random number between 20 and 250, which is recorded with string format.

In the energy smart contract, the function CreateAsset was defined with a third argument, energyUsage. The reason for this argument is to receive the power meter readings from the apartment.

A different node application triggers the energy smart contract. In it, the function updateAsset runs, calling the smart contract UpdateAsset to match the apartments using the ApartmentID attribute. However, one crucial element, the value in the attribute EnergyUsage, is fetched from the apartment smart contract to be recorded in the energy smart contract using the function GetEnergyUsage and sending as parameters the ApartmentID and the ZKPs off-chain-generated signal.

As mentioned above, GetEnergyUsage is a function defined in the energy smart contract which has the following functionalities: a) call QueryEnergyUsage from the apartment smart contract to know the power meter reading, b) record the start and c) end times before and after the call and d) calculate and e) record the difference between them (latency) in the asset.

On the other hand, the apartment smart contract-defined function QueryEnergyUsage first verifies that the received signal arrayArg proves that the apartment is what it says it is. Second, if authorised, return the asset where the energy usage is recorded.

## Zero Knowledge Proof

### Implementation

The protocol implementation was done using the library snarkjs.js (Iden3, 2024b), a JavaScript implementation of the zk-SNARK protocol that can use several schemes such as Groth16, Plonk and Fflonk protocols. Additionally, a domain-specific language (DSL) named Circom (Iden3, 2024a) was used for creating the ZKPs circuit. This compiler defines arithmetic circuits and constraints to portray computational calculations to be proven and verified without revealing the inputs.

Plonk, Groth16 and Fflonk verification keys and proofs were generated off-chain for this proof of concept, which has been named “the prototype”. Further, the witness was computed off-chain. Once those processes were ended, the verification keys and the proofs of each protocol were hard-coded to the Hyperledger Fabric smart contracts to be used in the transactions.

The process was similar for the three protocols (Circom, 2023). First, the arithmetic circuit using circom was defined. Second, the circuit is compiled to get a low-level representation (R1CS) using the following command:

$ circom Multiplier2.circom --r1cs --wasm --sym

Third, after compiling the circuit, the next step is computing the witness using the script witness\_calculator.js. The output of this execution is the binary file witness.wtns containing the computed signals and MultiplierX.r1cs having the defined circuit constraints. These files are mandatory for creating the proof.

Fourth, generate a trusted setup and get a zk-SNARK proof. The trusted setup has two steps, one independent of the circuit called phase one, the powers of tau, and another that is circuit-dependent, named phase two. By using the powers of tau ceremony, the trusted setup becomes more decentralised. The ceremony starts with the execution of the following command:

$ snarkjs powersoftau new bn128 12 pot12\_0000.ptau -v

In this case, the creation of the secret using the powers of tau ceremony allows multiple actors to contribute to the creation of the random secret.

After that comes the contribution to the ceremony:

$ snarkjs powersoftau contribute pot12\_0000.ptau pot12\_0001.ptau --name="First contribution" -v

The trusted setup of phase two starts with the execution of the following command:

$ snarkjs powersoftau prepare phase2 pot12\_0001.ptau pot12\_final.ptau -v

The function powersoftau included in the snarkjs library contains a series of functions related to the ceremony. Contribute and preparePhase2 are two of those. The first one, contribute, receives as parameters a) the existing powers of tau file, b) the new file that would be generated, c) the contribution id, d) a randomness value used in the contribution (entropy). Finally, e) an optional logger. Second, preparePhase2, receives as parameters a) the existing powers of tau file name, b) the new powers of tau file name and c) the optional logger. Both functions return a void promise.

Followed by the generation of the .zkey file containing the proving and keys for verification plus phase two contributions all together after the execution of the following command:

$ snarkjs fflonk setup Multiplier3.r1cs pot12\_final.ptau Multiplier3\_final.zkey

Once the .zkey file is created, the verification key was exported:

$ snarkjs zkey export verificationkey Multiplier3\_final.zkey verification\_key.json

Finally, create and validate the proof with the following commands:

$ snarkjs fflonk prove Multiplier3\_final.zkey witness.wtns proof.json public.json

$ snarkjs fflonk verify verification\_key.json public.json proof.json

### Circuit definition

This simple circuit is defined with Circom zksnark.js library to prove that the prime numbers multiplication plus the the cube of the result is known without revealing the number.

Firstly, input and output circuits were defined using two arbitrarily selected prime numbers (7,3), obtaining their product and the cube of that product (9261) as shown below in Figure 4.

A screen shot of a computer program

Description automatically generated

Figure 4. Circuit definition using circom for the prototype.

Additionally, the computational model for the proof of concept as shown in Figure 4. In this case, the verification consists of knowing the cube (9261) of the product without knowing the original factors (7,3). Finally, the public input expectedCube is returned and published as it is presented in Figure 13.

Set up is made by the powers of the tau ceremony, assuring it is random. The randomness of powers of tau is defined as a key in the zk-SNARK context (Ethereum, 2022). Its proposal complies with two requirements: first, creating a public parameter set to be used at the verification, also called secret randomness. Second, involving several rounds of contributions, where each round adds randomness to the parameter. This way, there is no single contributor controlling the outcome. The verification key, public signals and the proof will be used for verification.

To verify the proof, three files are passed as parameters: verification\_key.json, publicSignals.json and proof.json.

Using ZKPs to test that the secret is known, the proof and the verification keys created off-chain were manually added to the smart contract. This is not a normal process. However, it was done to simplify the validation process. This proof should have been sent from the back end in a production application.

As part of the same process, verification keys were added to energy and apartment smart contracts and appeared in Figure 14. Additionally, the file public.json, as its name suggests, contains the public input values for the circuit, called signals, as illustrated in Figure 15.

The file proof.json in Figure 16 contains the proof itself generated by snarkjs. These numbers, the cube of the result of the product of 7 x 3, are the signal that will be passed as a parameter.

To make the proof of concept easy to use, verification\_key and proof were copied into the Fabric inside the file assetTransfer.js and stored as variables.

Finally, the verifier method, which needed to verify the sender, was extended with the new function verify. The verification process uses three different functions depending on the protocol needed: plonk, groth16 or fflonk, the three of them being ZKPs system verification functionality from snarkjs, which receives the verification keys, the input and the proof as parameters.

In the three cases, as the generated keys were small, the off-chain process of generating them finished in less than a minute. As said above, this process uses the powers of the tau ceremony, a component that creates the public argument to be used in the cryptographic proof.

The file input.json displayed in Figure 17 contains the input values and the public signal’s value, whether public or private. In this case, 7 and 3 are private inputs, and 9261 is a public input, the signal to be sent for verification.

In both energy and apartment smart contracts, the Verify function was implemented for each protocol, Plonk, Groth16 and Fflonk. In this function, inputs are received as a parameter to validate that the secret is known and, most importantly, that the function snarkjs.plonk.verify can prove it. For example, if the secret is validated, the function running on the energy smart contract can read the power meter energy usage from the apartment smart contract, fetch that data and update its asset by recording the energy usage at the energy smart contract.

## SnarkJS installation set-up process

This research uses zk-SNARK for the validation, proofing, and key generation processes to validate the two different smart contracts. As Hyperledger Fabric doesn’t support this protocol by default, the JavaScript library SnarkJS was added to the blockchain (Szego, 2024).

The process of installing the library started with the command npm install snarkjs@latest. Second, by using the language functionality require to add modules, as shown in Figure 18, the SnarkJS library was added to the Hyperledger Fabric smart contracts, energy and apartment, to use the protocols, Plonk, Groth16 and Fflonk, all features provided by the library to verify the input data complies with the defined circuit.

Figure 19 reveals that the SnarkJS library was correctly added to the Fabric environment as a dependency, hence allowing the verification proof in the energy and apartment smart contracts. The smart-contracts configuration file package.json is shown on the above library as a dependency.

## Smart-contract Application

Running the prototype requires the setup of two smart contracts: apartment and energy.

### Apartment Smart Contract

#### CreateAsset

On the one hand, the apartment smart contract is responsible for creating apartments through the CreateAsset function as displayed in Figure 20. This function receives the context, AparmentID and energyUsage as parameters and creates assets with two attributes, ApartmentID and EnergyUsage.

#### QueryEnergyUsage

This function returns an asset depending on the ApartmentID passed as a parameter as exposed in Figure 21. To do so, the function calls the function Verify, which does the ZKPs verification of the signal passed as a parameter.

As the prototype was run with three different protocols, Plonk, Groth16 and Fflonk, three different functions were coded to comply with the requirement.

### Energy Smart Contract

#### CreateAsset

On the other hand, the energy smart contract also creates apartments that mirror those in the apartment smart contract. However, apartments in the energy smart contract will be created with the attribute EnergyUsage equal to 0, as it is shown in Figure 22. This value is meant to be updated by querying the energy usage for each apartment in the apartment smart contract.

#### GetEnergyUsage

The second function described in this smart contract is GetEnergyUsage Figure 23. This function is responsible for connecting both smart contracts. To do so requires using the function invokeChaincode, which receives the name of the smart contract as the first parameter, an array with the name of the apartment smart contract function to be used as well as the parameters that it would use, and the name of the channel where the smart contract resides as the third parameter.

### Common for both Smart Contract

Both smart contracts implement three different functions to verify the received request is from a valid source. Each of them uses a different protocol to do so, Plonk, Groth16 and Fflonk.

#### VerifyPlonk

VerifyPlonk function uses snarkjs.plonk.verify functionality from snarkjs library to validate the sender. It receives the verification keys, the signal and the proof as parameters, returning a Boolean value, as shown in Figure 24.

#### VerifyGroth16

VerifyGroth16 function uses snarkjs.groth16.verify functionality from snarkjs library to validate the sender. It receives the verification keys, the signal and the proof as parameters, returning a Boolean value, as shown in Figure 25.

#### VerifyFflonk

VerifyFflonk uses snarkjs.fflonk.verify functionality from snarkjs library to validate the sender. It receives the verification keys, the signal and the proof as parameters, returning a Boolean value, as shown in Figure 26.

## TypeScript Applications

### Apartment TypeScript App

#### createAsset

Once deployed, the smart contracts are ready to execute when called. In this prototype, two Node.js applications are responsible for calling them. The first application is responsible for telling the apartment smart contract how many apartments should be created by running its createAsset function, as shown in Figure 27.

Even though the typescript application createAsset function’s first responsibility is calling the apartment smart contract function CreateAsset to create the apartment, it also generates the ApartmentID and a random integer, used as the energy usage value, to be passed as parameters to initialise the apartment asset.

### Energy TypeScript App

#### createAsset

The first responsibility for this application is to create apartments for the energy smart contract, generating an ApartmentID for each apartment, which has to match the apartments in the apartment smart contract as shown in Figure 28.

#### updateAsset

Additionally, this app implemented the updateAsset function, having the responsibility to invoke the UpdateAsset in the energy smart contract. It also records the assets with more and less latency and calculates the latency average, as shown in Figure 29. Finally, it saves all the latency values in an array to be exported and analysed.

## The Oracle

Beniiche (2020) defines a blockchain oracle as a conceptualised representation mechanism enabling blockchain networks to interact with external data, called “off-chain” data. This data must originate from a credible source and must be accessible to the blockchain network oracles function as an Application Programming Interface (API) accessible via the public internet. They offer an endpoint that can be programmatically invoked from a Smart Contract, returning the requested data. The software hosting the required information can take various forms, while the API delivers a structured response in accordance with the user’s specifications. For example, a weather application on a smartphone may initiate an API request to a meteorological service, retrieving data in a specified format, which it subsequently processes to generate a visual representation. In the realm of smart contracts, the procedure is analogous; the contract requests data in a particular format and validates the received response based on its internal logic, thereby determining subsequent actions. In this prototype, the Oracle API functionality is defined at the energy node application.

Officiating as the Oracle API, the oracleUpdateAsset function is declared on the energy node application and calls the CallOracle function, exhibited in Figure 31, getting in return an Accuweather API object.

After fetching the object, oracleUpdateAsset calls the function OracleUpdateAsset declared in the energy smart contract exposed in Figure 32. The following arguments are sent: the ApartmentID, the Phrase and the TypeID retrieved from the Accuweather object.

# Prototype Execution

The planification of the testing was done considering twenty execution tests with ten requests, with the expected result of more than 0.65 requests per second. From the literature review, it is also expected that most of the time would happen during ZKPs generation.

However, in view of the massive amount of data that a smart city has to manage, the setup of the test was changed as follows.

## Explanation

The test was defined to run 10,000 times sequentially for each protocol to provide the ZKPs verification. The goal was to calculate the time needed to get the energy usage in the apartment smart contract from the energy smart contract.

To run the prototype, the apartments first have to be created. To do so, the code from the createAsset function is installed from the apartment node application using npm install. After the installation, the script is started using the command npm start. The terminal output of the createAsset function execution is displayed in Figure 33.

Even though Apartment0 is created and an EnergyUsage value assigned, its reading was not used in the prototype to fetch the energy usage from the energy smart contract.

After the apartment’s power meter creation (EnergyUsage), the energy node application will be installed and executed to create the apartments in the energy smart contract, following a similar process to the one executed for the apartments but from \_application-gateway-typescript-energy folder. First, the script installation using the npm install command. Second, start the script using npm start command to create the apartments with EnergyUsage equal to 0. The terminal output after running the script is shown in Figure 34.

The apartments represented in the energy smart contract will be updated immediately after its creation by fetching the energy usage from the apartment smart contract validating its identity by using ZKPs, with the execution of the energy smart contract UpdateAsset. The terminal output can be seen in Figure 35

As stated above, before triggering the function to retrieve the energy usage, a timer is started and finished when the data is retrieved. The difference is calculated, and the three pieces of information are recorded at the asset with the attribute names StartTime, EndTime and Latency.

## The Oracle

The OracleUpdateAsset declared at the energy smart contract uses the type ID attribute received as a parameter to decide the energy cost. On the one hand, when the API answers TypeID equals 0, it represents that it is not raining. It is assumed that hydroelectric energy generation would be more expensive (23 per energy usage unit). As it is shown in Figure 45, the retrieved data from Accuweather API, TypeID and Phrase, are recorded in the asset for clarification. In addition, the price and the calculated invoice are stored too.

On the other hand, when the API answers TypeID equals 1, it means it is raining or it will be in the next two hours, and the hydroelectric energy generation would be cheaper (19 per energy usage unit). Finally, all the values received and calculated are recorded in the asset as presented in Figure 46.

# Results and Discussion

## Results

## Plonk

The first test used the Plonk protocol to fetch the energy usage value from each apartment smart-contract power meter. After 10,000 queries, the average mean latency was 24.354 milliseconds per query. The lowest time to gather the data from the apartment smart contract was 17 milliseconds for Apartment 580, and the highest latency was for Apartment 9955, which spent 59 milliseconds running the query. The terminal output for both assets is shown in Figure 36 and Figure 37. The terminal output for the Plonk protocol summary test execution is presented in Figure 38.

## Groth16

The second out of three tests used Groth16 fetched the energy usage value from each apartment smart-contract power meter. After 10,000 queries, the average mean latency was 22.523 milliseconds per query. The lowest time to gather the data from the apartment smart contract was 16 milliseconds for Apartment 1963, and the highest latency was for Apartment 1, which ran the query for an extraordinary 394 milliseconds, both terminal outputs shown in Figure 39 and Figure 40. The terminal output for the Groth16 protocol summary test execution is presented in Figure 41.

## Fflonk

The last test used to fetch the energy usage value from each apartment smart-contract power meter was Fflonk. After 10,000 queries, the average mean latency was 20.5358 milliseconds per query. The lowest time to gather the data from the apartment smart contract was 12 milliseconds for Apartment 2092, and the highest latency was for Apartment 9986, which spent 44 milliseconds running the query. The terminal output for both assets is shown in Figure 42 and Figure 43. The terminal output for the Plonk protocol summary test execution is presented in Figure 44.

## Discussion

The results here show that ZKPs are a good way to verify the proofs. Fflonk, one of the protocols tested, turned out to be the fastest and most reliable, which means it can handle the constant flow of data in a smart city without reducing the speed or compromising privacy.

## Data Analysis

As presented in Figure 6, the three protocols tested in this dissertation show a relatively similar performance. It is worth expressing that in the case of the Plonk protocol, an outlier point of 59 milliseconds was left outside of the histogram. Utilising the same criteria, in the case of the Groth16 protocol, an outlier point of 394 milliseconds was also left outside of the graphic. The breakdown of the resulting test data can be found in the next sections.

Figure 6. Histogram of the three protocols by milliseconds.

### Plonk

The x-axis shows the test’s input size, 10,000 assets. The y-axis presents the measurement time needed to fetch the data after making the proof using the Plonk protocol represented in milliseconds.

As the variation in the y-axis is not significant when x-axis values increase, a significant variation is not expected if the input number increases, suggesting a stable performance of the protocol that is little related to the input value.

The cause of the presence of several points above the main cluster is not known. The lower value is 17, and the highest is 59 milliseconds However, due to the characteristics of the personal computer on which this test was run, they could indicate computational overhead.

Figure 7. Plonk protocol performance data after running 10,000 queries.

After looking at the result of the Plonk protocol test, even though the sample is just 10,000, it is confirmed that its verification time is efficient, as stated by (Ambrona et al., 2022).

Plonk protocol remains stable when the number of assets increases, avoiding significant performance degradation. Despite some peaks, the data distribution clustering in the y-axis is tight, suggesting good performance consistency with an average mean of 24.35 milliseconds, 24 milliseconds median, and 1.122 milliseconds standard deviation.

Figure 8. Plonk protocol performance data after running 10,000 queries grouped by milliseconds.

### Groth16

The Groth16 protocol requires a special mention of the first value of the test, an outstanding peak of 394 milliseconds, and a more expected lowest value of 16 milliseconds. Avoiding the first peak of the test, the analysis shows the x-axis input size of 10,000 assets. The y-axis presents the performance metric of the communication between energy and apartment smart contracts and the Groth16 protocol verification represented in milliseconds.

A stable performance of the protocol is suggested due to the little y-axis variation when the number of assets (x-axis) increases.

Figure 9. Groth16 protocol performance data after running 10,000 queries.

The performance of the Groth16 protocol verification is similar but more consistent to the Plonk protocol when the outlier is removed from the data set as shown in . However, although some outliers are present, they are less than in Plonk protocol verification, suggesting a more consistent performance. Furthermore, this suggests fewer exceptional cases affecting performance.

Like the Plonk protocol, Groth16 remains stable when the number of assets increases, avoiding significant performance degradation. However, the data distribution clustering in the y-axis is tighter, suggesting a better performance consistency when the first value of the test is avoided.

Finally, considering the notable 394 milliseconds first value of the test, that point suggests that some transactions can be significantly slower. In addition, Groth16 data collected after the test showed an average mean of 22.523 milliseconds, 22 milliseconds media, and 4.17411 standard deviation.

Figure 10. Groth16 protocol performance data after running 10,000 queries grouped by milliseconds.

The chart above, which groups energy assets by time used to fetch the energy usage from their correspondent apartment asset at the apartment smart contract, and verify the origin of the request using Groth16, shows that 28.82% of the assets needed 22 milliseconds and 67.82% of the assets required between 21 and 23 milliseconds.

### Fflonk

The input for Fflonk protocol was 10,000 assets, the same as the previous test of Plonk and Groth16 protocols. In this case, the points for the Fflonk protocol are densely clustered within the y-axis narrow range between 12 and 44 milliseconds, the time it takes to fetch the data and make the verification.

Figure 11. Fflonk protocol performance data after running 10,000 queries.

The test data suggest that this protocol has less propensity for variation than Plonk and Groth16 when increasing the input of assets. There are fewer outliers than Plonk and similar to Groth16 when excluding the outstanding value. However, Fflonk has the least amount of outliers, reinforcing its reliability.

Overall, displaying a lowest of 12 milliseconds, a highest record of 44 milliseconds, an average mean of 20.5358 milliseconds, and a standard deviation of 1.46565, Fflonk protocol showed the most stable performance of the three protocols.

Figure 12. Fflonk protocol performance data after running 10,000 queries grouped by milliseconds.

## Research Q1 answer

Q1: How can a distributed and agnostic mechanism be created to establish connections between different blockchain types to interconnect different smart contracts in a smart city?

As was shown in this dissertation, using ZKPs protocols would add a layer of privacy and security to enable the verification of the data between two smart contracts but without revealing the data. Making that work for connecting two smart contracts from different blockchains would improve its functionality, getting closer to the goal of blockchain interconnectivity in a smart city.

A different approach can be taken by using events, enabling connectivity between smart contracts from different blockchains using oracles. As the approach presented in the paragraph above, this one would need a layer of privacy and security provided by ZKPs protocols, doing the verifications in the smart contracts connected and managing the events in the off-chain applications. The data flow centralisation in the external application would negatively impact the distribution of the system and that would represent a risk and point of interest for potential intruders.

Finally, a new point of view is managing the connection between the smart contracts in different blockchains, using ZKPs protocols to add a layer of privacy and security, but instead of using a centralised off-chain application -such as the ones (energy and apartment nodeJS applications) used in the prototype for this dissertation- using a third blockchain to manage the events and connect the others blockchains.

## Research Q2 answer

Q2: How can ZKPs achieve cross-smart-contract, anonymous and low-latency transactions in a smart city context, and how are these attributes (anonymous, low-latency) related?

The three protocols tested, Groth16, Plonk, and Fflonk works having cross-smart-contract anonymous and low-latency transactions in smart cities are the principal attributes of the ZKPs protocol. It lets smart contracts share data securely without revealing private information.

Cross smart contract transactions in smart cities context need different systems, like energy meters and traffic lights, to share information securely. ZKPs let them do that without showing all the data. For example, the energy smart contract pulls data from the apartment smart contract without revealing private details about the energy usage. Groth16, which was the first protocol, is quick at verifying proofs but needs a “trusted setup,” which makes some people worried about security (Groth, 2016). After testing, Groth16 had an average latency of 22.523 milliseconds, making it faster than Plonk but with more variability in the results​.

ZKPs also keep transactions anonymous by hiding the actual data while still proving that it’s correct. For example, the apartment’s energy usage could be verified by the energy smart contract without revealing the actual meter readings. Plonk allows this kind of verification without showing all the information, and it only needs one setup for all tasks (Gabizon et al., 2019). In the testing done for this dissertation, Plonk had an average latency of 24.354 milliseconds, but it showed a bit more variability with some peaks reaching up to 59 milliseconds, which could slow things down​.

Fflonk was designed to be faster and handle larger workloads better. The following test shows that this protocol had the best performance with the lowest latency transactions, with an average latency of 20.535 milliseconds. Although it was the fastest protocol, its latency remained stable even with more transactions. This attribute becomes imperative in smart cities where systems like energy meters would have to operate in real-time. Fflonk’s, by definition, means fast. It also means transactions can happen quickly, keeping the city running smoothly while still maintaining privacy (Ambrona et al., 2022).

ZKPs keep the verification fast, in opposition of the fact that anonymous transactions usually reduce the speed of (Ernstberger et al., 2024; Gabizon et al., 2019). Low latency is a critical attribute to be considered in smart cities where decisions need to happen very fast. No revealing private information is another mandatory attribute to ponder, and ZKPs help with this as well. This balance is mandatory when implementing cross smart contract transactions. Different systems can trust each other’s data quickly without seeing all the details, keeping transactions fast and private.

## Research Q3 answer

Q3: What are the security considerations in a smart city when implementing cross smart contract communication?

There are security concerns when making different smart contracts interact with each other. Basically, what it means is that different parts of the city, smart devices, wearables, smart power meters, and servers to process the data and make decisions have the necessity of sharing data. Consequently, maintaining data safe from hackers is paramount.

First is data privacy. The code developed for the prototype of this dissertation shows a way in which the energy smart contract fetches the power meter readings from the apartment smart contract without exposing the acutal data, in this case, how much energy the apartment consumed. In the prototype, ZKPs protocol allowed the smart contract to validate the data but never disclosed the user’s sensitive data (Gabizon et al., 2019).

Second comes anonymity, generating trust. The desirable solution to maintain trust and anonymity presented by Fan et al. (2024) in a peer to peer (P2P) network scheme is using the “*key less*” ZKPs. Moreover, the authors established that the only thing that the verifier knows is the fact that the secret is true. Besides, as per the definition of non-interactive intrinsic to the protocol, the prover and the verifier cannot communicate with each other. In the proposed prototype for this dissertation, this characteristic can be seen in the functions: VerifyPlonk, VerifyGroth16 and VerifyFflonk, where only three parameters were received: the verification key, the signals and the proof.

# Conclusion

This dissertation was looking to test ZK protocols in a smart city context. The first contribution was that an application was developed to implement three different ZK methods, Groth16, Plonk and Fflonk. The second result is that Fflonk is faster and more consistent than the others by around two and four milliseconds. The final result is that there are security considerations.

Among the approaches tested, this dissertation found that Fflonk consistently offered the best performance, showing the lowest latency and the most reliable results. This suggests that Fflonk could be a solid choice for use cases that require fast, efficient communication between smart contracts, where speed is a critical factor.

There’s a lot more work to be done in this area. Making these ZKP protocols scale better and cutting down the latency even more could make them way more useful for bigger setups. Another thing worth digging into is how oracles, which pull in outside data, can be hooked into this system while still keeping the same level of privacy and security.

## Limitations

Even though the results look good, there are some obvious drawbacks. The first one is that the tests were all done on a local setup computer, which means the system might not perform in the same way in the real world, a much more complicated environment. Second, the tests were conducted with both smart contracts running on the same channel, so there is still much more to figure out when it comes to using these techniques across different blockchain networks, which is becoming more of a thing in actual practice. Finally, not having two or more blockchains connected is an evident limitation, as this could be useful for interconnection in a smart city.

## Further research

One possible step forward in this research would be using ZKPs to verify the communication between the NodeJS application officiating as the oracle and the API with the required data. However, the Accuweather API doesn’t allow modification to the API to add this feature.

It would also be desirable to connect different kinds of blockchains, for example, a permissioned blockchain like Fabric connected with a permissionless blockchain such as Ethereum. Furthermore, creating an agnostic hub where any blockchain type can connect with each other would be, at least, beneficial in a smart city context. However, anonymity, privacy, security and speed have to be considered. Finally, measuring the latency when the challenges of the circuit are scaling, or the scaling up state with ZKPs in the prototype with ZK rollups is also a fair possibility in the next steps of the investigation.

It is worth saying that for the three protocols tested, the time measured included the ZKPs verification and the time used by the smart contract energy to fetch the requested data from the smart contract apartment.

In the context of smart cities, where everything is connected such as traffic lights, energy meters and public services, you’ve got data flying around everywhere. The challenge is to make sure that data gets where it needs to go without exposing private information, like people’s energy usage or personal details.

Additionally, after the first approach, the following use case and components were defined:

Use case title: Smart Energy Grid based on the weather forecast (oracle) and blockchain (2) interoperability.

Chaincode (smart contract): Business logic to manage the business and its associated apartments. Includes electricity usage and invoice. The template used is “asset-transfer-basic/chaincode-javascript”.

Channel. All the apartments are in the same channel named “mychannel”.

Transaction. Are made by apartment. E.g., when a new transaction is made, the power consumption of the apartment or the new owner is updated, and the transaction is registered.

# References

Ahmid, M., & Kazar, O. (2023). A comprehensive review of the internet of things security. *Journal of Applied Security Research*, *18*(3), 289-305.

Ambrona, M., Schmitt, A.-L., Toledo, R. R., & Willems, D. (2022). New optimization techniques for PlonK’s arithmetization. *Cryptology ePrint Archive*.

Arogundade, O. R. (2023). Network security concepts, dangers, and defense best practical. *Computer Engineering and Intelligent Systems*, *14*(2).

Ballesteros-Rodríguez, A., Sánchez-Alonso, S., & Sicilia-Urbán, M.-Á. (2024). Enhancing Privacy and Integrity in Computing Services Provisioning Using Blockchain and zk-SNARKs. *IEEE Access*.

Beniiche, A. (2020). A study of blockchain oracles. *arXiv preprint arXiv:2004.07140*.

Bernabe, J. B., Canovas, J. L., Hernandez-Ramos, J. L., Moreno, R. T., & Skarmeta, A. (2019). Privacy-preserving solutions for blockchain: Review and challenges. *IEEE Access*, *7*, 164908-164940.

Buterin, V. (2014). A next-generation smart contract and decentralized application platform. *white paper*, *3*(37), 2-1.

Buterin, V. (2021). *An Incomplete Guide to Rollups*. <https://vitalik.eth.limo/general/2021/01/05/rollup.html>

Chen, Z., Jiang, Y., Song, X., & Chen, L. (2023). A survey on zero-knowledge authentication for internet of things. *Electronics*, *12*(5), 1145.

Circom. (2023). *Proving circuits with ZK*. <https://docs.circom.io/getting-started/proving-circuits/>

Dagdelen, Ö., Fischlin, M., & Gagliardoni, T. (2013). The Fiat–Shamir transformation in a quantum world. Advances in Cryptology-ASIACRYPT 2013: 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II 19,

Enisa. (2023). <https://www.enisa.europa.eu/topics/incident-response/glossary/blockchain>

Ernstberger, J., Chaliasos, S., Zhou, L., Jovanovic, P., & Gervais, A. (2024). Do You Need a Zero Knowledge Proof? *Cryptology ePrint Archive*.

Eskhita, R., Manda, V. K., & Hlali, A. (2021). Dubai and Barcelona as Smart Cities: Some Reflections on Data Protection Law and Privacy. *Environmental Policy and Law*, *51*(6), 403-407.

Ethereum. (2022). *Powers of tau specification*. <https://github.com/ethereum/kzg-ceremony-specs>

Fabric, H. (2022). *Secured asset transfer in Fabric*. <https://hyperledger-fabric.readthedocs.io/en/release-2.5/secured_asset_transfer/secured_private_asset_transfer_tutorial.html>

Fabric, H. (2023). *Fabric chaincode lifecycle*. <https://hyperledger-fabric.readthedocs.io/en/release-2.5/chaincode_lifecycle.html>

Fan, W., Wu, S., & Zou, Y. (2024). A Keyless Authentication Based on Zero-Knowledge Proof with SDN Link Information to Secure Permissionless P2P Networking. ICC 2024-IEEE International Conference on Communications,

Farayola, O. A., Olorunfemi, O. L., & Shoetan, P. O. (2024). Data privacy and security in IT: a review of techniques and challenges. *Computer Science & IT Research Journal*, *5*(3), 606-615.

Fiege, U., Fiat, A., & Shamir, A. (1987). Zero knowledge proofs of identity. Proceedings of the nineteenth annual ACM symposium on Theory of computing,

Gabizon, A., Williamson, Z. J., & Ciobotaru, O. (2019). Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*.

Groth, J. (2016). On the size of pairing-based non-interactive arguments. Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35,

Hasan, J. (2019). Overview and applications of zero knowledge proof (ZKP). *International Journal of Computer Science and Network*, *8*(5), 2277-5420.

Huang, C., & Shen, X. S. (2024). Decentralized Privacy Preservation in Smart Cities.

Iden3. (2024a). *Circom Circuit Compiler*. <https://docs.circom.io/>

Iden3. (2024b). *SnarkJs*. <https://github.com/iden3/snarkjs>

Javed, A. R., Ahmed, W., Alazab, M., Jalil, Z., Kifayat, K., & Gadekallu, T. R. (2022). A comprehensive survey on computer forensics: State-of-the-art, tools, techniques, challenges, and future directions. *IEEE Access*, *10*, 11065-11089.

Khan, K. M., Arshad, J., & Khan, M. M. (2020). Investigating performance constraints for blockchain based secure e-voting system. *Future Generation Computer Systems*, *105*, 13-26.

Liu, T., Xie, T., Zhang, J., Song, D., & Zhang, Y. (2024). Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. 2024 IEEE Symposium on Security and Privacy (SP),

Martinez, S., Ameigenda, A., De Barros, B., Llambias, G., González, L., & Ruggia, R. (2024). Leveraging Zero-Knowledge Proofs for Blockchain Interoperability: Experiences with Ethereum and Hyperledger Fabric. *Authorea Preprints*.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Satoshi Nakamoto*.

Oh, J., Yu, S., Lee, J., Son, S., Kim, M., & Park, Y. (2021). A secure and lightweight authentication protocol for IoT-based smart homes. *Sensors*, *21*(4), 1488.

Rao, P. M., & Deebak, B. D. (2023). Security and privacy issues in smart cities/industries: technologies, applications, and challenges. *Journal of Ambient Intelligence and Humanized Computing*, *14*(8), 10517-10553.

Singh, S. (2024). Enhancing Privacy and Security in Large-Language Models: A Zero-Knowledge Proof Approach. International Conference on Cyber Warfare and Security,

Szego, D. (2024). *Zero Knowledge Proof Workshop*. <https://github.com/Daniel-Szego/Zeroknowledgeworkshop>

Zarrin, J., Wen Phang, H., Babu Saheer, L., & Zarrin, B. (2021). Blockchain for decentralization of internet: prospects, trends, and challenges. *Cluster Computing*, *24*(4), 2841-2866.

Zhang, K., Ni, J., Yang, K., Liang, X., Ren, J., & Shen, X. S. (2017). Security and privacy in smart city applications: Challenges and solutions. *IEEE communications magazine*, *55*(1), 122-129.

Zhang, Y., Qu, Y., Gao, L., Luan, T. H., Jolfaei, A., & Zheng, J. X. (2023). Privacy-preserving data analytics for smart decision-making energy systems in sustainable smart community. *Sustainable Energy Technologies and Assessments*, *57*, 103144.

Zhou, L., Diro, A., Saini, A., Kaisar, S., & Hiep, P. C. (2024). Leveraging zero knowledge proofs for blockchain-based identity sharing: A survey of advancements, challenges and opportunities. *Journal of Information Security and Applications*, *80*, 103678.

# Acknowledgements

This dissertation would not have come to life without the support of Maria, Sofia, Maite, Federica and Florencia. They are my family and have the patience to support me when I am tired and always encourage me to keep going forward.

It is also an act of justice to acknowledge my supervisors Shu and Jeff. This job wouldn’t have been possible without their support, compromise and wisdom.

# Appendix

## The circuit



Figure 13. The expected cube is exposed to release the pubic signals.

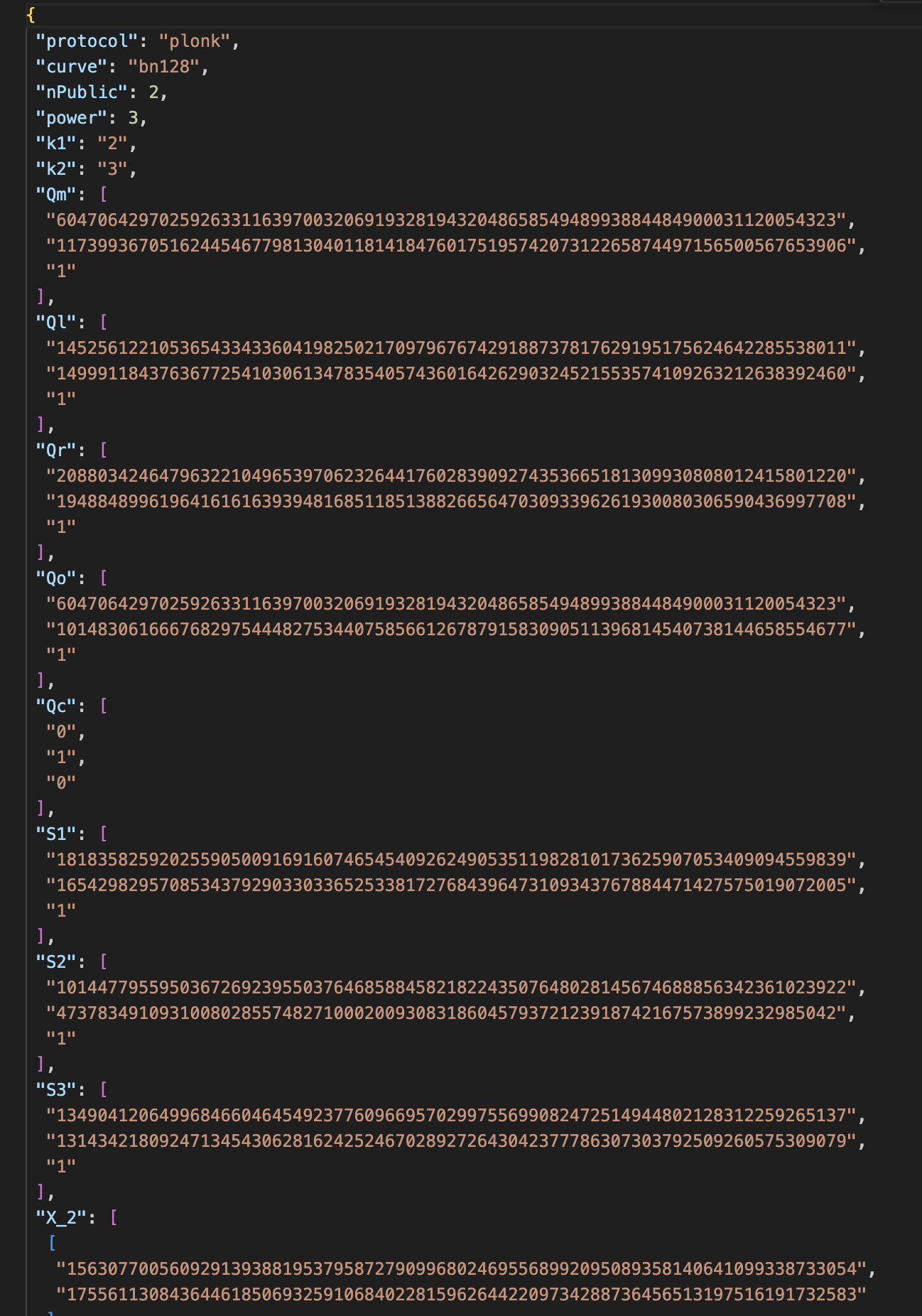


Figure 14. Verification key output. This value was hard coded in the smart contract for ZKPs verification.

A screenshot of a computer

Description automatically generated

Figure 15. File public.json with the public signal values.

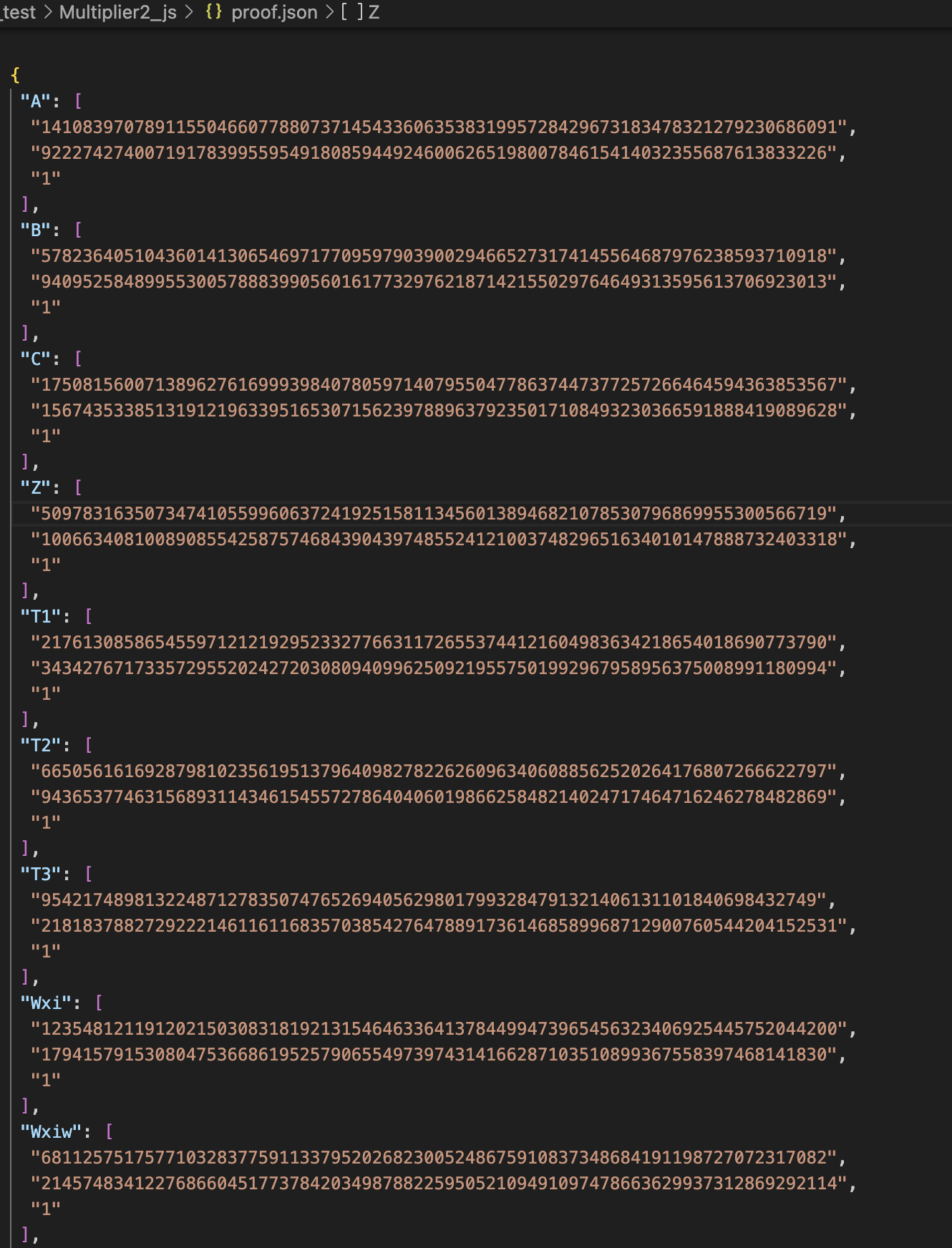


Figure 16. File proof.json generated storing the proof value to be used at the smart contracts for ZKPs verification.

A screenshot of a computer program

Description automatically generated

Figure 17. File input.json with the public and private inputs.

## SnarkJS

A screen shot of a computer

Description automatically generated

Figure 18. Require JavaScript functionality to include a module in the code.

A screenshot of a black screen

Description automatically generated

Figure 19. The energy and apartment smart-contracts configuration file package.json is shown with snarkjs library added as a dependency.

## Smart contracts



Figure 20. Energy smart contract CreateAsset defines the apartment structure.



Figure 21. Apartment smart contract function QueryEnergyUsage returns the energy used in each apartment.

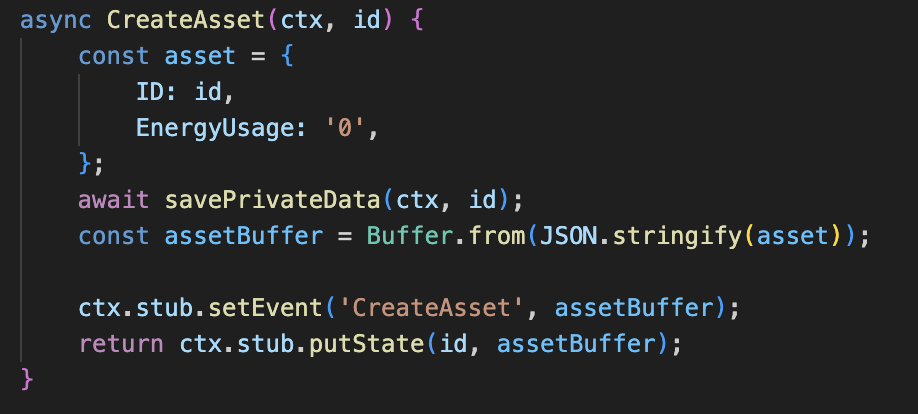


Figure 22. CreateAsset function in Energy smart contract.

A screen shot of a computer program

Description automatically generated

Figure 23. GetEnergyUsage in energy smart contract.

A screen shot of a computer code

Description automatically generated

Figure 24. VerifyPlonk function code.

A computer screen with text

Description automatically generated

Figure 25. VerifyGroth16 function code.

A screen shot of a computer code

Description automatically generated

Figure 26. VerifyFflonk function code.

## Node applications

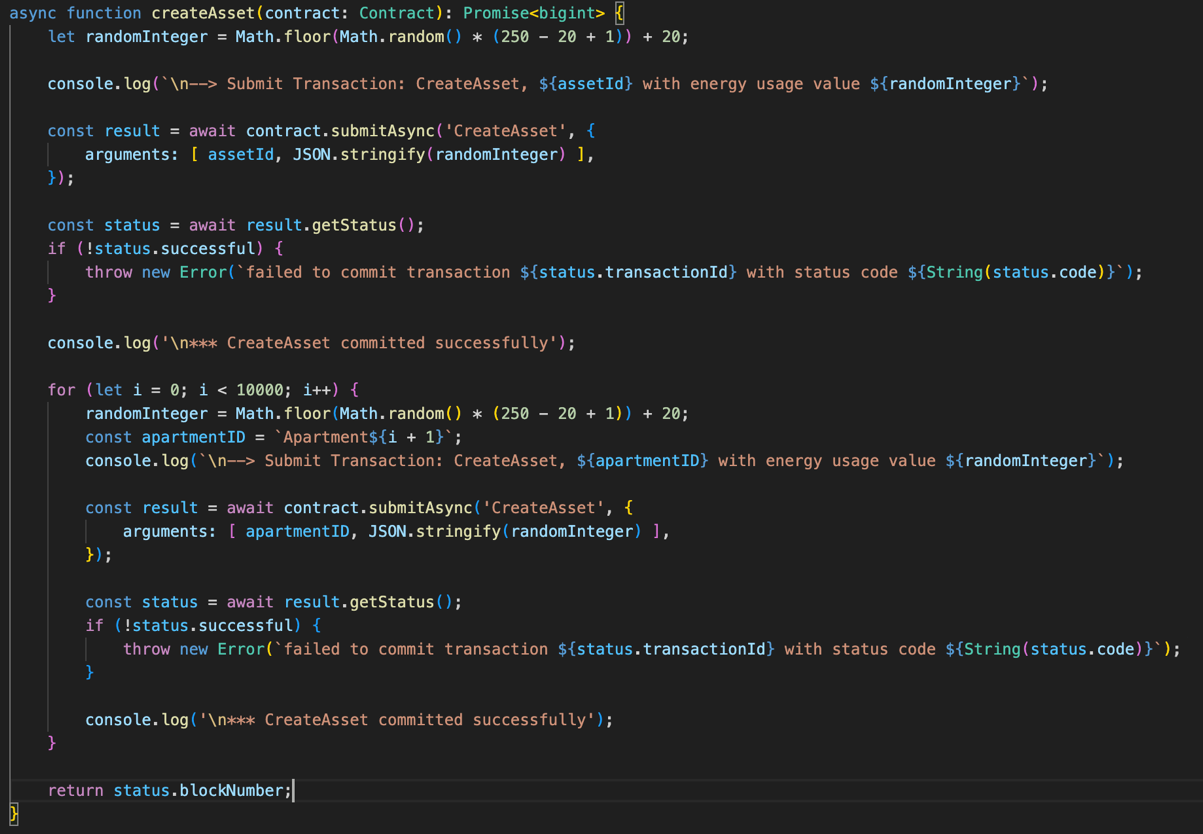


Figure 27. NodeJS application createAsset function for creating apartments with energy usage random integer value in apartment smart contract.



Figure 28. NodeJS application createAsset function for creating apartments with energy usage value equal 0 in energy smart contract.

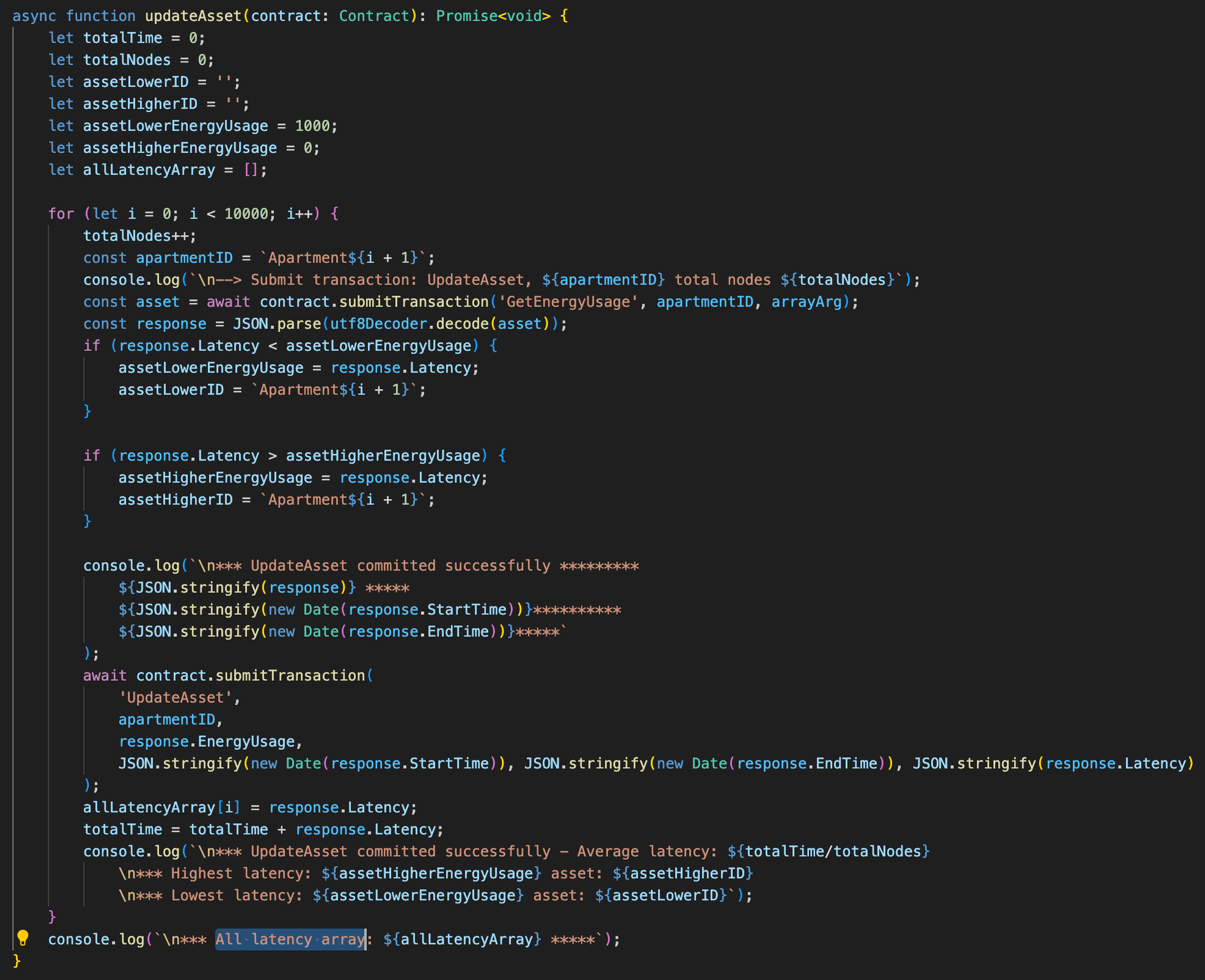


Figure 29. Energy NodeJS application updateAsset triggers the process of fetching energy usage from apartment smart contract.

A screen shot of a computer program

Description automatically generated

Figure 30. Energy NodeJS application oracleUpdateAsset function.

A screen shot of a computer

Description automatically generated

Figure 31. Energy NodeJS application CallOracle function.

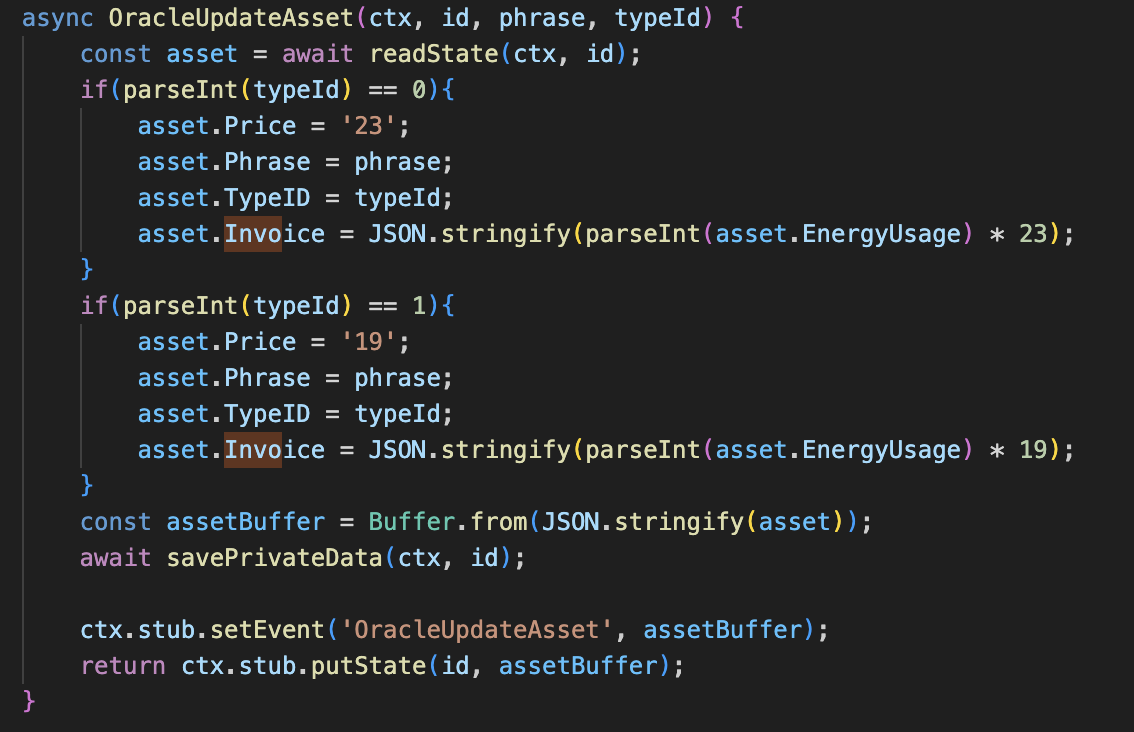


Figure 32. Energy smart contract OracleUpdateAsset function.

## ZKPs Prototype execution

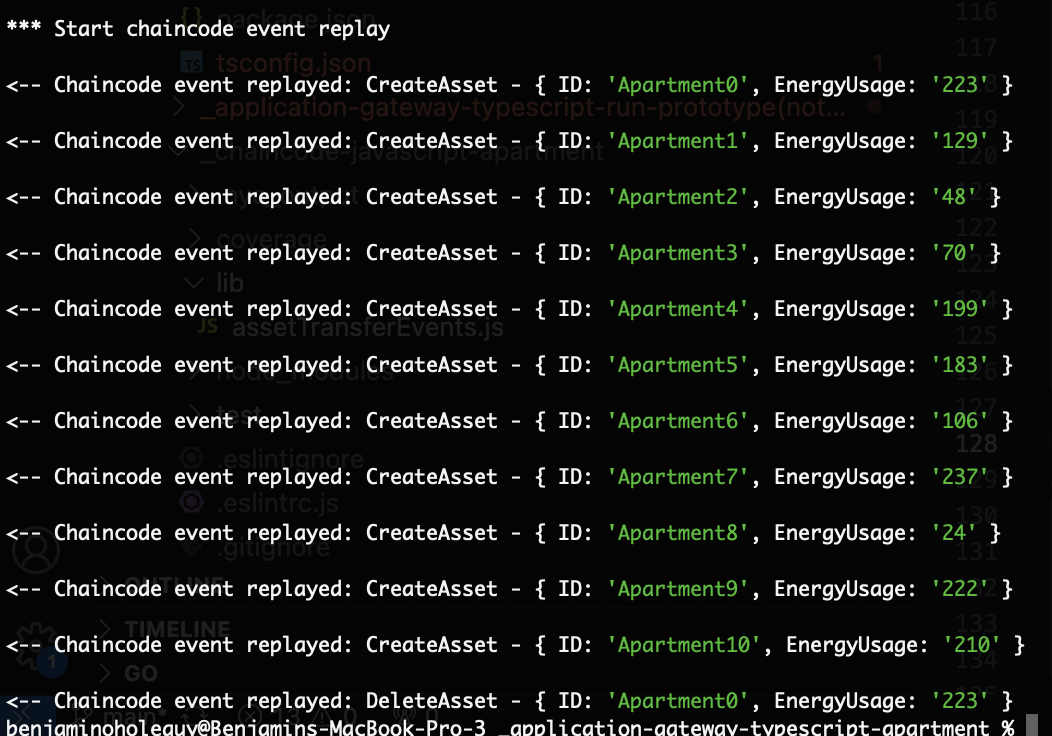


Figure 33. Apartment smart contract CreateAsset function terminal output.



Figure 34. Energy smart contract function CreateAsset with EnergyUsage attribute equals 0.

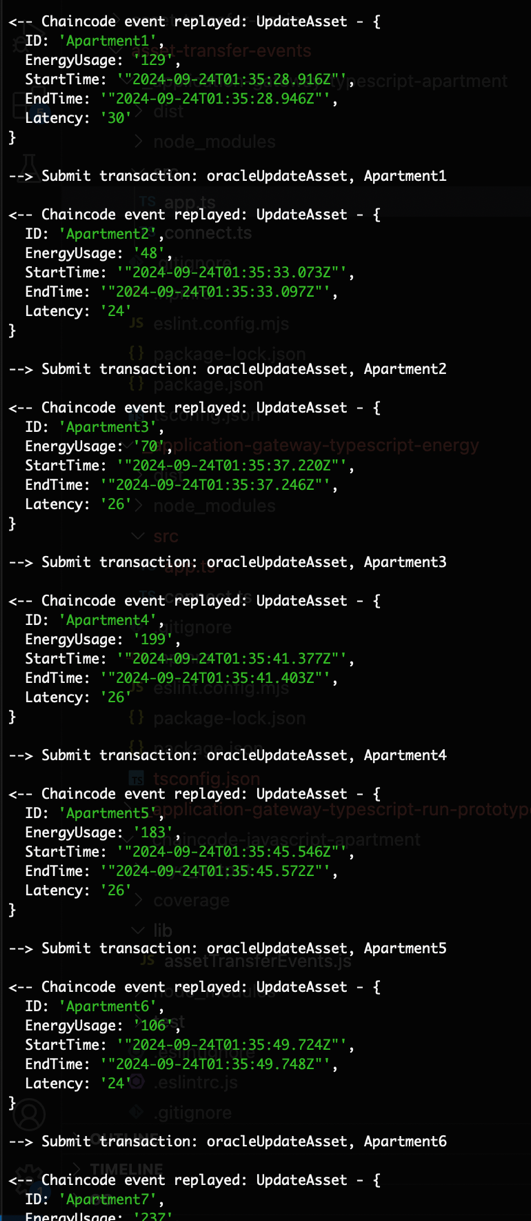


Figure 35. Energy smart contract UpdateAsset function output.

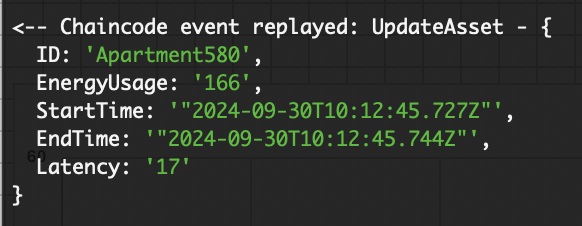


Figure 36. Energy smart contract UpdateAsset terminal output execution showing the asset with the lowest latency verifying with the Plonk protocol.

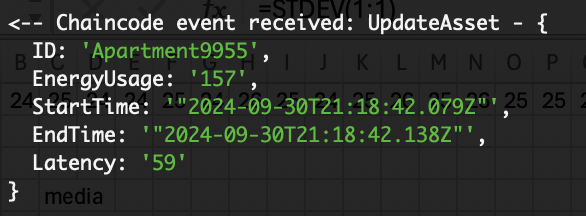


Figure 37. Energy smart contract UpdateAsset terminal output execution showing the asset with the highest latency verifying with the Plonk protocol.

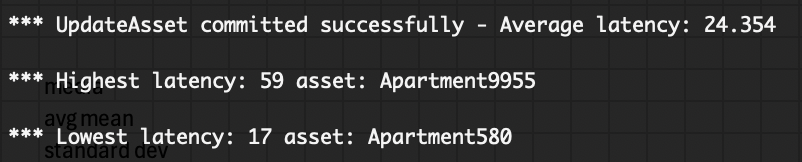


Figure 38. Energy smart contract UpdateAsset terminal output summary using the Plonk protocol.

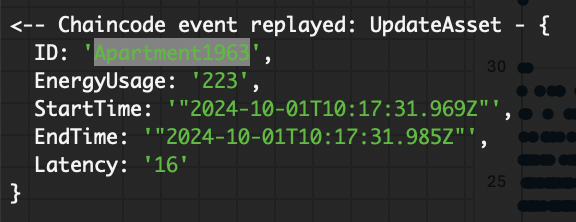


Figure 39. Energy smart contract UpdateAsset terminal output execution showing the asset with the lowest latency verifying with the Groth16 protocol.

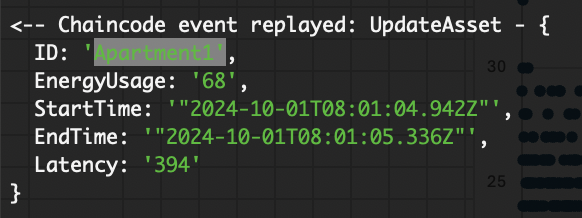


Figure 40. Energy smart contract UpdateAsset terminal output execution showing the asset with the highest latency verifying with the Groth16 protocol.

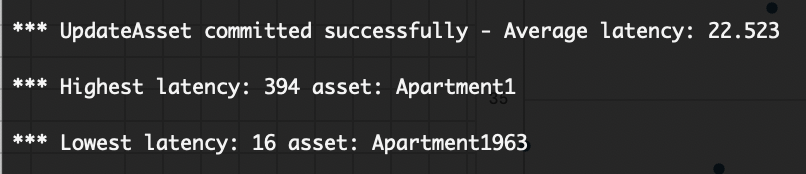


Figure 41. Energy smart contract UpdateAsset terminal output summary using the Groth16 protocol.

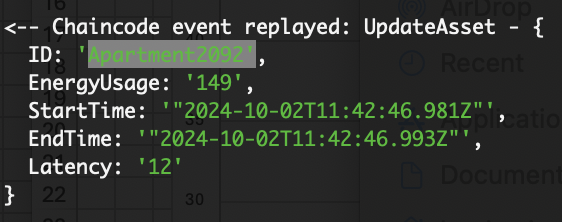


Figure 42. Energy smart contract UpdateAsset terminal output execution showing the asset with the lowest latency verifying with the Fflonk protocol.

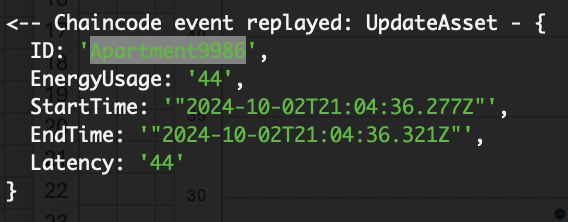


Figure 43. Energy smart contract UpdateAsset terminal output execution showing the asset with the highest latency verifying with the Fflonk protocol.

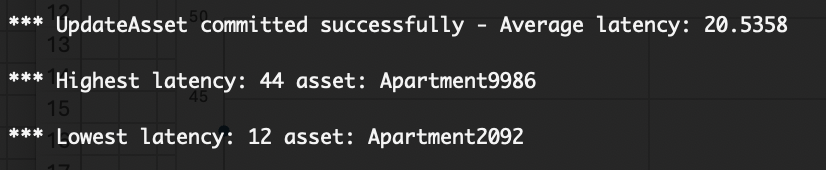


Figure 44. Energy smart contract UpdateAsset terminal output summary using the Fflonk protocol.

## The oracle. Prototype execution

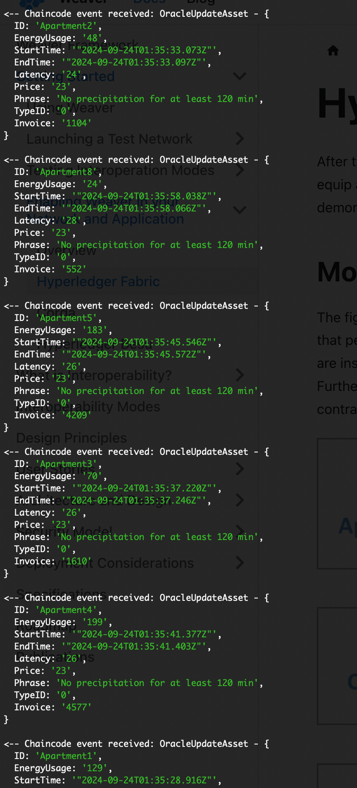


Figure 45. OracleUpdateAsset terminal output execution with Accuweather API attribute TypeID equals 0.

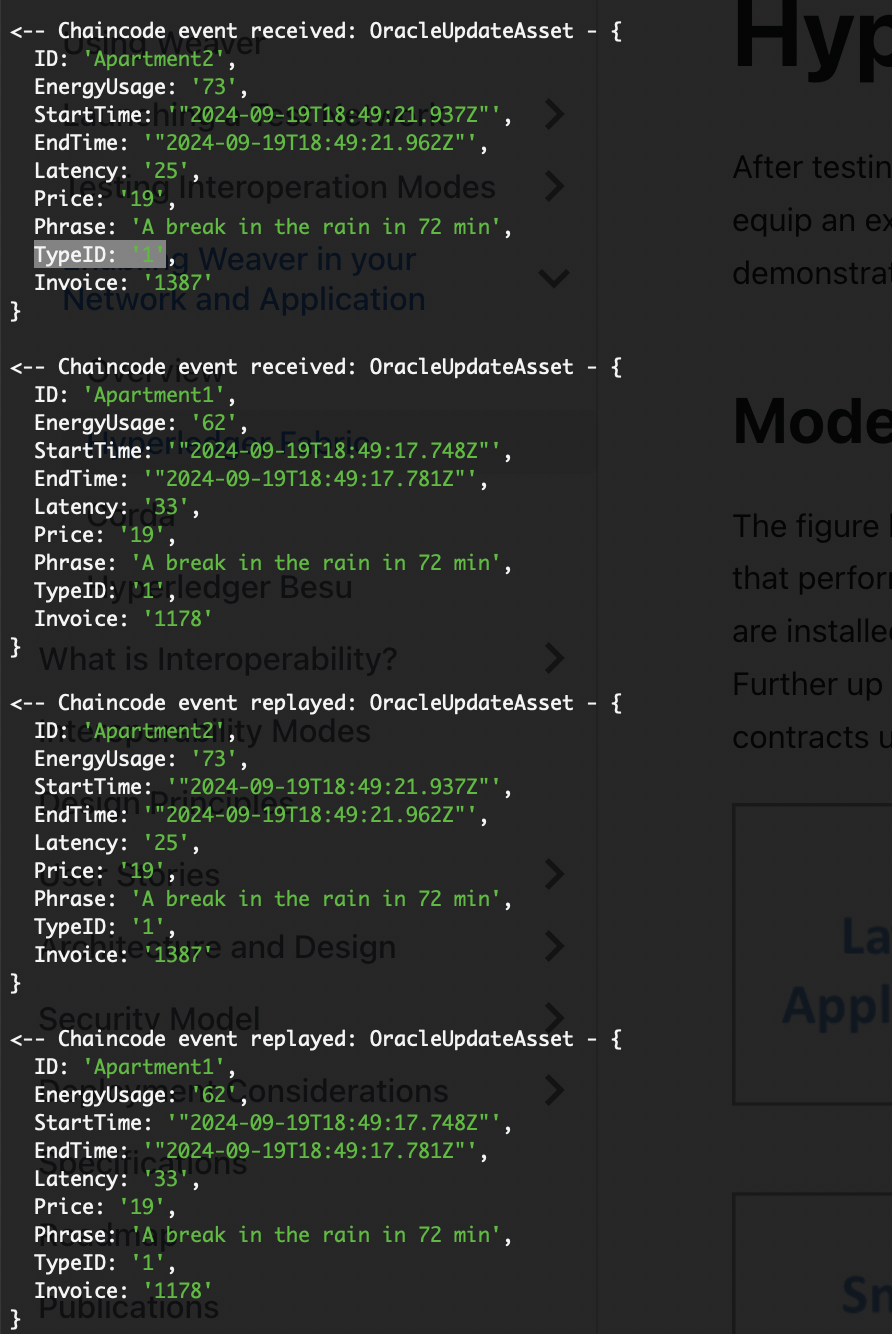


Figure 46. OracleUpdateAsset terminal output execution with Accuweather API attribute TypeID equals 1.

1. https://github.com/benjaoholeguydeveloper/dissertation [↑](#footnote-ref-1)